Lecture 4:
Neural Networks and Backpropagation
Announcements

Cloud credits for projects: we are in the process of securing them and will announce them as soon as we can.

Assignment 1 due Fri 4/19 at 11:59pm
Administrative: Project Proposal

Due Mon 4/22

TA expertise are posted on the webpage.

(http://cs231n.stanford.edu/office_hours.html)
Administrative: Live Q&A

For students who are watching the lecture online live:

- We are hosting a live Q&A session on Ed
- Questions will be responded to by TAs as much as possible.
- See the Live Lecture Q&A megathread pinned on Ed for more information
Administrative: Discussion Section

Discussion section tomorrow (led by Lucas Leanza):

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

```python
# 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
```
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:** \[ \alpha_t = \frac{\alpha_0}{\sqrt{t}} \]

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

Deep Learning
“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.
In a fantastical setting, a highly detailed furry humanoid skunk with piercing eyes confidently poses in a medium shot, wearing an animal hide jacket. The artist has masterfully rendered the character in digital art, capturing the intricate details of fur and clothing texture.

An illustration from a graphic novel. A bustling city street under the shine of a full moon. The sidewalks bustling with pedestrians enjoying the nightlife. At the corner stall, a young woman with fiery red hair, dressed in a signature velvet cloak, is haggling with the grumpy old vendor. The grumpy vendor, a tall, sophisticated man wearing a sharp suit, who sports a noteworthy mustache is animatedly conversing on his steampunk telephone.

User: What is unusual about this image?

GPT-4: The unusual thing about this image is that a man is ironing clothes on an ironing board attached to the roof of a moving taxi.

User: Can you explain this meme?

GPT-4: This meme is a joke that combines two unrelated things: pictures of the earth from space and chicken nuggets. The text of the meme suggests that the image below is a beautiful picture of the earth from space. However, the image is actually of chicken nuggets arranged to vaguely resemble a map of the world. The humor in this meme comes from the unexpected juxtaposition of the text and the image. The text sets up an expectation of a majestic image of the earth, but the image is actually something mundane and silly.

Image source: https://openai.com/research/gpt-4
Segment Anything Model (SAM)
https://openai.com/research/video-generation/models-as-world-simulators
Sora

- Animating Images (generated by DALL-E)
- Video-to-video editing

A Shiba Inu dog wearing a beret and black turtleneck.

put the video in space with a rainbow road

change the video setting to be different than a mountain? perhaps joshua tree

https://openai.com/research/video-generation-models-as-world-simulators
Sora

- More compute

https://openai.com/research/video-generation-models-as-world-simulators
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)
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
\[ f = W_2 \max(0, W_1x) \]
\[ x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H} \]

“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

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

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

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

(Now) 2-layer Neural Network
\[ f = W_2 \max(0, W_1x) \]

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?

\[ f = W_2 W_1 x \]

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

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

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

\[ \tanh(x) \]

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

ReLU is a good default choice for most problems

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

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

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

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

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

“Fully-connected” layers
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)
ggrad2 = h.T.dot(grad_y_pred)
ggrad_h = grad_y_pred.dot(w2.T)
ggrad_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)
ggrad_w2 = h.T.dot(grad_y_pred)
ggrad_h = grad_y_pred.dot(w2.T)
ggrad_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)

TensorFlow Play Ground: https://playground.tensorflow.org/
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

dendrite

presynaptic terminal

axon

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**

**dendrite**

**cell body**

**axon**

**presynaptic terminal**

**sigmoid activation function**

\[
\frac{1}{1 + e^{-x}}
\]
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”, IEEE/CVF International Conference on Computer Vision 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

\[ s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \] \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}
Problem: How to compute gradients?

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

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

$$s = f(x; W) = Wx$$

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

$$= \sum_{j \neq y_i} \text{max}(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_{k} W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \text{max}(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_{k} W_k^2$$

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

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

\[ f = W x \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]
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.
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. \( x = -2, y = 5, z = -4 \)

\[
\begin{align*}
q &= x + y \\
\frac{\partial q}{\partial x} &= 1, \quad \frac{\partial q}{\partial y} = 1
\end{align*}
\]
Backpropagation: a simple example

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

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

\[
\begin{align*}
q &= x + y \\
\frac{\partial q}{\partial x} &= 1, \quad \frac{\partial q}{\partial y} = 1
\end{align*}
\]

\[
\begin{align*}
f &= qz \\
\frac{\partial f}{\partial q} &= z, \quad \frac{\partial f}{\partial z} = q
\end{align*}
\]
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: \(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\)
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. \( x = -2, y = 5, z = -4 \)

<table>
<thead>
<tr>
<th>( q = x + y )</th>
<th>( \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f = qz )</td>
<td>( \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q )</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
q &= x + y & \frac{\partial q}{\partial x} &= 1, & \frac{\partial q}{\partial y} &= 1 \\
\end{align*}
\]

\[
\begin{align*}
f &= qz & \frac{\partial f}{\partial q} &= z, & \frac{\partial f}{\partial z} &= q \\
\end{align*}
\]

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

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

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

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
q &= x + y \\
\frac{\partial q}{\partial x} &= 1, \quad \frac{\partial q}{\partial y} = 1
\end{align*}
\]

\[
\begin{align*}
f &= qz \\
\frac{\partial f}{\partial q} &= z, \quad \frac{\partial f}{\partial z} = q
\end{align*}
\]

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

Chain rule:

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

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
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:

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

Upstream gradient

Local gradient
"local gradient"
"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 relationship between gradients in a function $f$. The notation $\frac{\partial L}{\partial z}$, $\frac{\partial z}{\partial x}$, and $\frac{\partial z}{\partial y}$ represent the derivatives of the loss $L$ with respect to $z$, and the partial derivatives of $z$ with respect to $x$ and $y$, respectively.

- **Downstream gradients** refer to $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$.
- **Local gradient** is represented by $\frac{\partial L}{\partial x}$.
- **Upstream gradient** is represented by $\frac{\partial L}{\partial z}$.
\[ \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x} \]

"Downstream gradients"

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

"Upstream gradient"

\[ \text{"local gradient"} \]

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

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

"local gradient"

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

"Upstream gradient"
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:

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

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

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

\[
\begin{align*}
  f(x) &= e^x &\rightarrow& &\frac{df}{dx} &= e^x \\
  f_a(x) &= ax &\rightarrow& &\frac{df}{dx} &= a
\end{align*}
\]

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

Upstream gradient \( (1.00) \left( \frac{-1}{1.37^2} \right) = -0.53 \)

Local gradient
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:

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

\[
\begin{align*}
\text{Upstream gradient} & \quad \text{Local gradient} \\
(-0.53)(1) & \quad = -0.53
\end{align*}
\]

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

\[
\begin{align*}
f_a(x) &= ax \\
\frac{df}{dx} &= a
\end{align*}
\]

\[
\begin{align*}
f(x) &= \frac{1}{x} \\
\frac{df}{dx} &= -\frac{1}{x^2}
\end{align*}
\]

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

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

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

Upstream gradient  Local gradient

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

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

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

\[ \text{Upstream gradient} \quad (\text{Local gradient}) \quad (-0.20)(-1) = 0.20 \]

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

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

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

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

\[
[0.2] \times [1] = 0.2
\]

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

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

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

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

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

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

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

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

[upstream gradient] \times [local gradient]

\begin{align*}
w_0 & : [0.2] \times [-1] = -0.2 \\
x_0 & : [0.2] \times [2] = 0.4
\end{align*}

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

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

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

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

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:

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

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!

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

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

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

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

Sigmoid local gradient:

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

[upstream gradient] x [local gradient]

\[
[1.00] \times [(1 - 0.73) (0.73)] = 0.2
\]
Patterns in gradient flow

add gate: gradient distributor

\[ \begin{array}{ccc}
3 & 2 & 4 \\
2 & 7 & 2 \\
\end{array} \]
Patterns in gradient flow

add gate: gradient distributor

mul gate: “swap multiplier”

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

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

add gate: gradient distributor

mul gate: “swap multiplier”

copy gate: gradient adder
Patterns in gradient flow

add gate: gradient distributor

mul gate: “swap multiplier”

copy gate: gradient adder

max gate: gradient router
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

```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
```
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_x0 = grad_s0 * x0
grad_w0 = 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
```

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

Multiply gate

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:

```python
# receive W (weights), X (data)
# forward pass (we have 8 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)
Example: PyTorch operators
PyTorch sigmoid layer

Forward

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

Source

#ifndef TH_GENERIC_FILE
#define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
#else

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_NELEMENT(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. - z) * z;
    );
}
#endif
The sigmoid layer in PyTorch is defined as:

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

This formula is used for the forward pass. The actual definition of the forward pass is provided in the code snippet:

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

This code snippet shows the implementation of the sigmoid function in PyTorch.
PyTorch sigmoid layer

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

Forward

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

Backward

Forward actually defined elsewhere...

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

<table>
<thead>
<tr>
<th>Scalar to Scalar</th>
<th>Vector to Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in \mathbb{R}, y \in \mathbb{R}$</td>
<td>$x \in \mathbb{R}^N, y \in \mathbb{R}$</td>
</tr>
<tr>
<td>Regular derivative: $\frac{\partial y}{\partial x} \in \mathbb{R}$</td>
<td>Derivative is Gradient: $\frac{\partial y}{\partial x} \in \mathbb{R}^N, \left(\frac{\partial y}{\partial x}\right)_n = \frac{\partial y}{\partial x_n}$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Scalar to Scalar</th>
<th>Vector to Scalar</th>
<th>Vector to Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in \mathbb{R}$, $y \in \mathbb{R}$</td>
<td>$x \in \mathbb{R}^N$, $y \in \mathbb{R}$</td>
<td>$x \in \mathbb{R}^N$, $y \in \mathbb{R}^M$</td>
</tr>
<tr>
<td>Regular derivative:</td>
<td>Derivative is Gradient:</td>
<td>Derivative is Jacobian:</td>
</tr>
<tr>
<td>$\frac{\partial y}{\partial x} \in \mathbb{R}$</td>
<td>$\frac{\partial y}{\partial x} \in \mathbb{R}^N$</td>
<td>$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M}$</td>
</tr>
<tr>
<td>If $x$ changes by a small amount, how much will $y$ change?</td>
<td>For each element of $x$, if it changes by a small amount then how much will $y$ change?</td>
<td>For each element of $x$, if it changes by a small amount then how much will each element of $y$ change?</td>
</tr>
</tbody>
</table>
Backprop with Vectors

Loss $L$ still a scalar!
Backprop with Vectors

\[ \mathbf{x}, \mathbf{y}, \mathbf{z} \]

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

$D_x \ x$

$D_y \ y$

$f$

$D_z \ z$

$\nabla L / \nabla z$

"Upstream gradient"

Loss $L$ still a scalar!
Backprop with Vectors

Loss $L$ still a scalar!

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

$D_x \ x$

$D_y \ y$

$\frac{\partial L}{\partial z}$

$D_z$

$z$

$f$

$D_z$
Backprop with Vectors

Loss $L$ still a scalar!

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

"Downstream gradients"

"Upstream gradient"

"local gradients"

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

"local gradients"

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

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

\[ [D_x \times D_z] \]

Jacobian matrices

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

"Downstream gradients"

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

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

\[ [D_y \times D_z] \]

Loss \( L \) still a scalar!

"Upstream gradient"

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

\[ D_z \]
Backprop with Vectors

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

Matrix-vector multiply

Jacobian matrices

"local gradients"

Loss \( L \) still a scalar!

"Downstream gradients"

"Upstream gradient"
Gradients of variables wrt loss have same dims as the original variable.

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

Loss L still a scalar!
Backprop with Vectors

<table>
<thead>
<tr>
<th>4D input x:</th>
<th>4D output z:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>[1]</td>
</tr>
<tr>
<td>[-2]</td>
<td>[0]</td>
</tr>
<tr>
<td>[3]</td>
<td>[3]</td>
</tr>
<tr>
<td>[-1]</td>
<td>[0]</td>
</tr>
</tbody>
</table>

\[ f(x) = \max(0, x) \] (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 $x$:

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

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

4D output $z$:

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

Jacobian $\frac{d z}{d x}$:

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

4D $\frac{d L}{d z}$:

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

Upstream gradient
Backprop with Vectors

4D input $x$:

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

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

4D output $z$:

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

\[
\begin{bmatrix}
dz/dx & dL/dz
\end{bmatrix}
\]

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

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 $dL/dx$:

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

4D $dL/dz$:

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

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

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

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

Upstream gradient
Backprop with Vectors

4D input x:

| 1 | -2 | 3 | -1 |

4D output z:

| 1 | 0 | 3 | 0 |

4D dL/dx:

| 4 | 0 | 5 | 0 |

4D dL/dz:

| 4 | -1 | 5 | 9 |

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

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 \( dL/dx \):

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

4D \( dL/dz \):

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

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

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

Upstream gradient
Backprop with Matrices (or Tensors)

\[
[D_x \times M_x] \quad x \quad \frac{dL}{dx}
\]

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

Matrix-vector multiply

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

Jacobian matrices

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

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

"Downstream gradients"

Matrix-vector multiply

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

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

"Upstream gradient"

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

\[ \frac{dL}{dx} \text{ always has the same shape as } x! \]

Loss \( L \) still a scalar!
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 \( L \)?

\[
[D_x \times M_x] \\
[D_y \times M_y] \\
[D_z \times M_z]
\]

“Downstream gradients”

“Upstream gradient”

Matrix-vector multiply

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

“local gradients”

Jacobian matrices

Loss \( L \) still a scalar!

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

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

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

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

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

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

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

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

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

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

\[
x: [N \times D] \\
[2 \ 1 \ -3] \\
[-3 \ 4 \ 2]
\]

\[
w: [D \times M] \\
[3 \ 2 \ 1 \ -1] \\
[2 \ 1 \ 3 \ 2] \\
[3 \ 2 \ 1 \ -2]
\]

\[
\text{Matrix Multiply}
\]

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

\[
y: [N \times M] \\
[13 \ 9 \ -2 \ -6] \\
[5 \ 2 \ 17 \ 1]
\]

\[
dL/dy: [N \times M] \\
[2 \ 3 \ -3 \ 9] \\
[-8 \ 1 \ 4 \ 6]
\]

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 \(~256\ \text{GB}\) of memory!

Must work with them implicitly!
Backprop with Matrices

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

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

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

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

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

\begin{align*}
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}
\end{align*}

\text{Matrix Multiply}
\begin{align*}
y_{n,m} &= \sum_d x_{n,d} w_{d,m} \\
n \in [N] & \quad m \in [M] \\
d \in [D]
\end{align*}

\begin{align*}
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}
\end{align*}

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

Q: How much does $x_{n,d}$ affect $y_{n,m}$?

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

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 \(y_n,\).

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

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

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

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

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

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

\[ \frac{\partial L}{\partial y_{n,m}}: \begin{bmatrix} N \times M \end{bmatrix} \]
\[
\begin{bmatrix}
2 & 3 & -3 & 9 \\
-8 & 1 & 4 & 6
\end{bmatrix}
\]

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: \( w_{d,m} \)

\[ \frac{\partial L}{\partial x} = \left( \frac{\partial L}{\partial y} \right)^T w \]
\[ \frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m} \]
Backprop with Matrices

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

Matrix Multiply

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

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

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

By similar logic:

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

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

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!