Lecture 30: Neural Nets and CNNs
Today

• PA 4 due this week
• PA 5 out this week. Due on Monday May 4
• HW 2 due next week
• Class on Monday (Charter Day)
Where are we?

1. Score function
   \[ f(x_i, W, b) = W x_i + b \]

2. Loss function
   \[ L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[ \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda R(W) \]
Where are we?

• Have score function and loss function
  – Will generalize the score function
• Find $W$ and $b$ to minimize loss
Where are we?

• Gradient descent to optimize loss functions
  – Batch gradient descent, stochastic gradient descent
  – Momentum
• Where do we get gradients of loss?
  – Compute them
  – Backpropagation and chain rule (more today)
- classification
- localization
- detection
- segmentation
Image classification: training

Training Images

Training Labels
Image classification: training

Training Images

Training Labels
Image classification: training

Training Images

Training Labels

Classifier Training
Image classification: training

Training Images

Training Labels

Classifier Training

Trained Classifier
Image classification: training

Training Images

Training Labels

Classifier Training

Trained Classifier
Image classification: testing

Test Image
Image classification: testing

Test Image
Image classification: testing

Test Image → Trained Classifier
Image classification: testing

Test Image ➔ Trained Classifier ➔ Prediction Outdoor
Traditional Recognition Approach

Input data (pixels) → feature representation (hand-crafted) → Learning Algorithm (e.g., SVM)

Image → Low-level vision features (edges, SIFT, HOG, etc.) → Object detection / classification

Slide: R. Fergus
Traditional Recognition Approach

Features are not learned

Input data (pixels) → feature representation (hand-crafted) → Learning Algorithm (e.g., SVM)

Image → Low-level vision features (edges, SIFT, HOG, etc.) → Object detection / classification
Computer vision features

SIFT

Spin image

HoG

Textons

and many others:

SURF, MSER, LBP, Color-SIFT, Color histogram, GLOH, ....

Slide: R. Fergus
Mid-Level Representations

- Mid-level cues
  - Continuation
  - Parallelism
  - Junctions
  - Corners

“Tokens” from Vision by D. Marr:

- Object parts:
  - Wheel
  - Eye
  - Nose
  - Eye with eyebrow

- Difficult to hand-engineer \(\rightarrow\) What about learning them?
Learning Feature Hierarchy

- Learn hierarchy
- All the way from pixels → classifier
- One layer extracts features from output of previous layer
Feature hierarchy with CNNs
End-to-end models

[Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", 2013]
IMAGENET Large Scale Visual Recognition Challenge

Year 2010

NEC-UIUC

Dense grid descriptor: HOG, LBP
Coding: local coordinate, super-vector
Pooling, SPM
Linear SVM

[Lin CVPR 2011]

Year 2012

SuperVision

Year 2014

GoogleLeNet

VGG

MSRA

[Simonyan arxiv 2014]

[He arxiv 2014]
Recent history of object detection

- **Large improvements** using Deep Learning [Girshick’13/14]

---


Convolutional Neural Networks

CS 4670
Sean Bell

When a user takes a photo, the app should check whether they're in a national park...
Sure, easy GIS lookup. Gimme a few hours.
...and check whether the photo is of a bird.
I'll need a research team and five years.

In CS, it can be hard to explain the difference between the easy and the virtually impossible.

(Sep 2014)
Introducing: Flickr PARK or BIRD

Zion National Park Utah by Les Haines (CC BY)

Secretary Bird by Bill Gracey (CC BY-NC-ND)
To play, drag an image from the examples or from your desktop.

PARK or BIRD

Want to know if your photo is from a U.S. national park? Want to know if it contains a bird? Just drag it into the box to the left, and we'll tell you. We'll use the GPS embedded in your photo (if it's there) to see whether it's from a park, and we'll use our super-cool computer vision skills to try to see whether it's a bird (which is a hard problem, but we do a pretty good job at it).

To try it out, just drag any photo from your desktop into the upload box, or try dragging any of our example images. We'll give you your answers below!

Want to know more about PARK or BIRD, including why the heck we did this? Just click here for more info →

PARK? BIRD?

Photo credits
PARK or BIRD

Want to know if your photo is from a U.S. national park? Want to know if it contains a bird? Just drag it into the box to the left, and we'll tell you. We'll use the GPS embedded in your photo (if it's there) to see whether it's from a park, and we'll use our super-cool computer vision skills to try to see whether it's a bird (which is a hard problem, but we do a pretty good job at it).

To try it out, just drag any photo from your desktop into the upload box, or try dragging any of our example images. We'll give you your answers below!

Want to know more about PARK or BIRD, including why the heck we did this? Just click here for more info ➔

EXAMPLE PHOTOS

PARK? YES

Ah yes, Bryce Canyon is truly beautiful.

Photo credits

BIRD? NO

Beautiful clouds, but I don't see any birds flying up there.
PARK or BIRD

Want to know if your photo is from a U.S. national park? Want to know if it contains a bird? Just drag it into the box to the left, and we'll tell you. We'll use the GPS embedded in your photo (if it's there) to see whether it's from a park, and we'll use our super-cool computer vision skills to try to see whether it's a bird (which is a hard problem, but we do a pretty good job at it).

To try it out, just drag any photo from your desktop into the upload box, or try dragging any of our example images. We'll give you your answers below!

Want to know more about PARK or BIRD, including why the heck we did this? Just click here for more info → 🌐

EXAMPLE PHOTOS

PARK?
YES
Hey, yeah! I went to Everglades once!

BIRD?
YES
Hey! Nice bird shot!

Photo credits
This week, we’ll learn what this is, how to compute it, and how to learn it.
What is a Convolutional Neural Network (CNN)?
What is a Convolutional Neural Network (CNN)?
What is a Convolutional Neural Network (CNN)?
What is a Convolutional Neural Network (CNN)?
What is a Convolutional Neural Network (CNN)?

Key questions:
What is a Convolutional Neural Network (CNN)?

Key questions:

- What kinds of functions should we use?
What is a Convolutional Neural Network (CNN)?

Key questions:
- What kinds of functions should we use?
- How do we learn the parameters for those functions?
Example CNN

Conv -> ReLU -> Conv -> ReLU -> Pool -> Fully Connected -> Softmax

*This network is running live in your browser*

[Andrej Karpathy]
CNNs in 1989: “LeNet”

LeNet: a classifier for handwritten digits. [LeCun 1989]
CNNs in 2012: “SuperVision” (aka “AlexNet”)

“AlexNet” — Won the ILSVRC2012 Challenge

Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

[Krizhevsky, Sutskever, Hinton. NIPS 2012]
CNNs in 2012: “SuperVision” (aka “AlexNet”)

“AlexNet” — Won the ILSVRC2012 Challenge

**Major breakthrough:** 15.3% Top-5 error on ILSVRC2012 (Next best: 25.7%)

Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

[Krizhevsky, Sutskever, Hinton. NIPS 2012]
CNNs in 2014: "GoogLeNet"

"GoogLeNet" — Won the ILSVRC2014 Challenge

[ Szegedy et al, arXiv 2014 ]
CNNs in 2014: “GoogLeNet”

“GoogLeNet” — Won the ILSVRC2014 Challenge

6.67% top-5 error rate!
(1000 classes!)

[Szegedy et al, arXiv 2014]
CNNs in 2014: “VGGNet”

“VGGNet” — Second Place in the ILSVRC2014 Challenge

No fancy picture, sorry

[Simonyan et al, arXiv 2014]
CNNs in 2014: “VGGNet”

“VGGNet” — Second Place in the ILSVRC2014 Challenge

[Simonyan et al, arXiv 2014]
CNNs in 2014: “VGGNet”

“VGGNet” — Second Place in the ILSVRC2014 Challenge

No fancy picture, sorry

7.3% top-5 error rate

(and 1st place in the detection challenge)

[Simonyan et al, arXiv 2014]
Neural Networks

(First we’ll cover Neural Nets, then build up to Convolutional Neural Nets)
Inspiration from Biology
Inspiration from Biology

A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Figure: Andrej Karpathy
Inspiration from Biology

Neural nets are loosely inspired by biology

Figure: Andrej Karpathy
Inspiration from Biology

Neural nets are **loosely inspired** by biology

But they certainly are **not** a model of how the brain works, or even how neurons work.

Figure: Andrej Karpathy
Let’s consider a simple 1-layer network:

\[ x \rightarrow Wx + b \rightarrow f \]
Simple Neural Net: 1 Layer

Let’s consider a simple 1-layer network:

\[ x \rightarrow Wx + b \rightarrow f \]

This is the same as what you saw last class:
1 Layer Neural Net

Block Diagram:

\[ x \rightarrow Wx + b \rightarrow f \]

(Input) \quad (class scores)
1 Layer Neural Net

Block Diagram:

\[ x \rightarrow Wx + b \rightarrow f \]

(Input) (class scores)

Expanded Block Diagram:

\[
\begin{bmatrix}
W \\
D
\end{bmatrix}
\begin{bmatrix}
x \\
D \\
1
\end{bmatrix}
+ \begin{bmatrix}
b \\
1
\end{bmatrix}
= \begin{bmatrix}
f \\
1
\end{bmatrix}
\]

M classes
D features
1 example
1 Layer Neural Net

Block Diagram:

\[ x \rightarrow Wx + b \rightarrow f \]

(Input) \hspace{2cm} (class scores)

Expanded Block Diagram:

\[
\begin{bmatrix}
M \\
D \\
1
\end{bmatrix}
\begin{bmatrix}
W \\
x \\
1
\end{bmatrix}
+ 
\begin{bmatrix}
M \\
1
\end{bmatrix}
= 
\begin{bmatrix}
f \\
1
\end{bmatrix}
\]

NumPy:

\[
f = np.dot(W, x) + b
\]

M classes

D features

1 example
1 Layer Neural Net
1 Layer Neural Net

• How do we process $N$ inputs at once?
1 Layer Neural Net

• How do we process $N$ inputs at once?

• It’s most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything
1 Layer Neural Net

• How do we process $N$ inputs at once?

• It’s most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything

$$X$$
1 Layer Neural Net

• How do we process $N$ inputs at once?

• It’s most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything
1 Layer Neural Net

• How do we process $N$ inputs at once?

• It’s most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything
1 Layer Neural Net

- How do we process $N$ inputs at once?

- It’s most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything.

$$X \times W + b = F$$

$N$ examples

$D$ features

$M$ classes

$N \times D$

$M \times D$

$1 \times 1$

$M \times M$

$M \times N$

$M \times N$
1 Layer Neural Net

- How do we process $N$ inputs at once?

- It’s most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything.

$$X \quad W \quad + \quad b \quad = \quad F$$

Note: Often, if the weights are transposed, they are still called “$W$”
1 Layer Neural Net
1 Layer Neural Net

Each row is one input example
1 Layer Neural Net

Each row is one input example

Each column is the weights for one class
1 Layer Neural Net

Each row is one input example

Each column is the weights for one class

Each row is the predicted scores for one example
1 Layer Neural Net

Implementing this with NumPy:

First attempt — let’s try this:

$$ F = \text{np.dot}(X, W) + b $$
1 Layer Neural Net

Implementing this with NumPy:

First attempt — let’s try this:

\[ F = \text{np.dot}(X, W) + b \]

Doesn’t work — why?
1 Layer Neural Net

Implementing this with NumPy:

First attempt — let’s try this:

\[ F = \text{np.dot}(X, W) + b \]

Doesn’t work — why?
1 Layer Neural Net

Implementing this with NumPy:

First attempt — let’s try this:

\[ F = \text{np.dot}(X, W) + b \]

Doesn’t work — why?

- NumPy needs to know how to expand “b” from 1D to 2D
1 Layer Neural Net

Implementing this with NumPy:

First attempt — let’s try this:

\[ F = \text{np.dot}(X, W) + b \]

Doesn’t work — why?

- NumPy needs to know how to expand “b” from 1D to 2D
- This is called “broadcasting”
1 Layer Neural Net

Implementing this with NumPy:

\[ F = \text{np.dot}(X, W) + b[\text{np.newaxis}, :] \]

What does “np.newaxis” do?
1 Layer Neural Net

Implementing this with NumPy:

\[ F = \text{np.dot}(X, W) + b[\text{np.newaxis}, :] \]

What does “np.newaxis” do?

```python
In [3]: b
Out[3]: array([0, 1, 2])
```

\[ b = [0, 1, 2] \]
1 Layer Neural Net

Implementing this with NumPy:

\[ F = \text{np.dot}(X, W) + b[np.newaxis, :] \]

What does “np.newaxis” do?

```
In [3]: b
Out[3]: array([0, 1, 2])
In [4]: b[np.newaxis, :]
Out[4]: array([[0, 1, 2]])
```

\[ b = [0, 1, 2] \]

Make “b” a row vector
1 Layer Neural Net

Implementing this with NumPy:

\[ F = \text{np.dot}(X, W) + b[\text{np.newaxis}, :] \]

What does “np.newaxis” do?

```python
In [3]: b
Out[3]: array([0, 1, 2])

In [4]: b[np.newaxis, :]
Out[4]: array([[0, 1, 2]])

In [5]: b[:, np.newaxis]
Out[5]: array([[0],
              [1],
              [2]])
```

b = [0, 1, 2]

Make “b” a row vector

Make “b” a column vector
1 Layer Neural Net

Implementing this with NumPy:

\[ F = \text{np.dot}(X, W) + b[\text{np.newaxis}, :] \]

What does “np.newaxis” do?
1 Layer Neural Net

Implementing this with NumPy:

\[ F = \text{np.dot}(X, W) + b[\text{np.newaxis}, :] \]

What does “np.newaxis” do?

```
In [12]: b[np.newaxis, :] + np.zeros((3, 3))
Out[12]:
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  2.],
       [ 0.,  1.,  2.]])
```

Row vector (repeat along rows)
1 Layer Neural Net

Implementing this with NumPy:

\[ F = \text{np.dot}(X, W) + b[\text{np.newaxis}, :] \]

What does “np.newaxis” do?

```
In [12]: b[np.newaxis, :] + np.zeros((3, 3))
Out[12]:
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  2.],
       [ 0.,  1.,  2.]])
```

Row vector (repeat along rows)

```
In [13]: b[:, np.newaxis] + np.zeros((3, 3))
Out[13]:
array([[ 0.,  0.,  0.],
       [ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

Column vector (repeat along columns)
2 Layer Neural Net

What if we just added another layer?

\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow h \]
2 Layer Neural Net

What if we just added another layer?

\[
 x \rightarrow W^{(1)}x + b^{(1)} \rightarrow h \rightarrow W^{(2)}h + b^{(2)} \rightarrow f
\]
2 Layer Neural Net

What if we just added another layer?

\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow h \rightarrow W^{(2)}h + b^{(2)} \rightarrow f \]

Let’s expand out the equation:
2 Layer Neural Net

What if we just added another layer?

\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow h \rightarrow W^{(2)}h + b^{(2)} \rightarrow f \]

Let’s expand out the equation:

\[
f = W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)}
= (W^{(2)}W^{(1)})x + (W^{(2)}b^{(1)} + b^{(2)})
\]
What if we just added another layer?

\[ f = W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)} \]

Let's expand out the equation:

\[ f = (W^{(2)}W^{(1)})x + (W^{(2)}b^{(1)} + b^{(2)}) \]

But this is just the same as a 1 layer net with:
2 Layer Neural Net

What if we just added another layer?

Let's expand out the equation:

\[ f = W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)} \]

\[ = (W^{(2)}W^{(1)})x + (W^{(2)}b^{(1)} + b^{(2)}) \]

But this is just the same as a 1 layer net with:

\[ W = W^{(2)}W^{(1)} \quad b = W^{(2)}b^{(1)} + b^{(2)} \]
2 Layer Neural Net

What if we just added another layer?

\[
x \rightarrow W^{(1)}x + b^{(1)} \rightarrow h \rightarrow W^{(2)}h + b^{(2)} \rightarrow f
\]

Let's expand out the equation:

\[
f = W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)} = (W^{(2)}W^{(1)})x + (W^{(2)}b^{(1)} + b^{(2)})
\]

But this is just the same as a 1 layer net with:

\[
W = W^{(2)}W^{(1)} \quad b = W^{(2)}b^{(1)} + b^{(2)}
\]

We need a **non-linear** operation between the layers
Nonlinearities
Nonlinearities

Sigmoid

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

Sigmoid

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

Tanh
Nonlinearities

Sigmoid

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

Tanh

ReLU
Nonlinearities

Historically popular

2 Big problems:
- Not zero centered
- They saturate
Nonlinearities

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

Historically popular

2 Big problems:
- Not zero centered
- They saturate

Tanh

- Zero-centered,
- But also saturates

ReLU
Nonlinearities

Sigmoid

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

Historically popular

2 Big problems:
- Not zero centered
- They saturate

Tanh

- Zero-centered,
- But also saturates

ReLU

- No saturation
- Very efficient
Nonlinearities

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

2 Big problems:
- Not zero centered
- They saturate

**Tanh**
- Zero-centered,
- But also saturates

**ReLU**
- No saturation
- Very efficient

Best in practice for classification
Nonlinearities — Saturation

What happens if we reach this part?

Sigmoid

In [22]: sigmoid = lambda x: 1 / (1 + np.exp(-x))
Nonlinearities — Saturation

What happens if we reach this part?

Sigmoid

In [22]: sigmoid = lambda x: 1 / (1 + np.exp(-x))

In [24]: sigmoid(np.array([-1, 2, 5]))
Out[24]: array([ 0.26894142, 0.88079708, 0.99330715])
Nonlinearities — Saturation

What happens if we reach this part?

**Sigmoid**

In [22]: sigmoid = lambda x: 1 / (1 + np.exp(-x))

In [24]: sigmoid(np.array([-1, 2, 5]))
Out[24]: array([ 0.26894142, 0.88079708, 0.99330715])

In [25]: sigmoid(np.array([-1, 2, 50]))
Out[25]: array([ 0.26894142, 0.88079708, 1.0])
Nonlinearities — Saturation

What happens if we reach this part?

Sigmoid

In [22]: sigmoid = lambda x: 1 / (1 + np.exp(-x))

In [24]: sigmoid(np.array([-1, 2, 5]))
Out[24]: array([ 0.26894142,  0.88079708,  0.99330715])

In [25]: sigmoid(np.array([-1, 2, 50]))
Out[25]: array([ 0.26894142,  0.88079708,  1.0])

In [26]: sigmoid(np.array([100, 200, 50]))
Out[26]: array([ 1.,  1.,  1.])
Nonlinearities — Saturation

What happens if we reach this part?

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

What happens if we reach this part?

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

\[ \frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x)) \]
Nonlinearities — Saturation

What happens if we reach this part?

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

\[ \frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x)) \]

dsigmoid = lambda x: sigmoid(x) * (1 - sigmoid(x))
Nonlinearities — Saturation

What happens if we reach this part?

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

dsigmoid = lambda x: sigmoid(x) * (1 - sigmoid(x))

In [29]: dsigmoid(np.array([-1, 2, 5]))
Out[29]: array([ 0.19661193, 0.10499359, 0.00664806])
Nonlinearities — Saturation

What happens if we reach this part?

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

\[
\text{dsigmoid} = \lambda x: \text{sigmoid}(x) \times (1 - \text{sigmoid}(x))
\]

In [29]: dsigmoid(np.array([-1, 2, 5]))
Out[29]: array([ 0.19661193,  0.10499359,  0.00664806])

In [30]: dsigmoid(np.array([100, 200, 50]))
Out[30]: array([ 0.,  0.,  0.])
Nonlinearities — Saturation

What happens if we reach this part?

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

\[
\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))
\]

dsigmoid = lambda x: sigmoid(x) * (1 - sigmoid(x))

In [29]: dsigmoid(np.array([-1, 2, 5]))
Out[29]: array([ 0.19661193,  0.10499359,  0.00664806])

In [30]: dsigmoid(np.array([100, 200, 50]))
Out[30]: array([ 0.,  0.,  0.])

Saturation: the gradient is zero!
Nonlinearities

[Krizhevsky 2012] (AlexNet)
Nonlinearities

In practice, ReLU converges ~6x faster than Tanh for classification problems.

[Krizhevsky 2012] (AlexNet)
ReLU in NumPy

Many ways to write ReLU — these are all equivalent:
ReLU in NumPy

Many ways to write ReLU — these are all equivalent:

(a) Elementwise max, “0” gets broadcasted to match the size of h1:

\[ h1_{\text{relu}} = \text{np.maximum}(h1, 0) \]
ReLU in NumPy

Many ways to write ReLU — these are all equivalent:

(a) Elementwise max, “0” gets broadcasted to match the size of h1:

```python
h1relu = np.maximum(h1, 0)
```

(b) Make a boolean mask where negative values are True, and then set those entries in h1 to 0:

```python
h1relu = h1.copy()
h1relu[h1 < 0] = 0
```
ReLU in NumPy

Many ways to write ReLU — these are all equivalent:

(a) Elementwise max, “0” gets broadcasted to match the size of h1:

\[
\text{h1relu} = \text{np.maximum}(\text{h1}, 0)
\]

(b) Make a boolean mask where negative values are True, and then set those entries in h1 to 0:

\[
\text{h1relu} = \text{h1.copy()}
\]
\[
\text{h1relu}[\text{h1} < 0] = 0
\]

(c) Make a boolean mask where positive values are True, and then do an elementwise multiplication (since \(\text{int(True)} = 1\)):

\[
\text{h1relu} = \text{h1} \times (\text{h1} \geq 0)
\]
2 Layer Neural Net
2 Layer Neural Net

\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow \]

(Layer 1)
2 Layer Neural Net

\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow \text{max}(0, \cdot) \rightarrow h^{(1)} \]

(Layer 1)    (Nonlinearity)    (“hidden activations”)
2 Layer Neural Net

\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow \max(0, \cdot) \rightarrow h^{(1)} \rightarrow W^{(2)}x + b^{(2)} \rightarrow f \]

(Layer 1) (Nonlinearity) (Layer 2)

(“hidden activations”)
Let's expand out the equation:

\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow \max(0, \cdot) \rightarrow h^{(1)} \rightarrow W^{(2)}x + b^{(2)} \rightarrow f \]

("hidden activations")
Let's expand out the equation:

\[ f = W^{(2)} \max(0, W^{(1)} x + b^{(1)}) + b^{(2)} \]
Let's expand out the equation:

\[ f = W^{(2)} \max(0,W^{(1)}x + b^{(1)}) + b^{(2)} \]

Now it no longer simplifies — yay
2 Layer Neural Net

Let's expand out the equation:

\[ f = W^{(2)} \max(0, W^{(1)} x + b^{(1)}) + b^{(2)} \]

Now it no longer simplifies — yay

**Note:** *any* nonlinear function will prevent this collapse, but not all nonlinear functions actually work in practice
2 Layer Neural Net

\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow \max(0, \cdot) \rightarrow h^{(1)} \rightarrow W^{(2)}x + b^{(2)} \rightarrow f \]

"hidden activations"

**Note:** Traditionally, the nonlinearity was considered part of the layer and is called an "activation function"

In this class, we will consider them separate layers, but be aware that many others consider them part of the layer.
Neural Net:
Graphical Representation
Neural Net: Graphical Representation

2 layers
Neural Net: Graphical Representation

- Called “fully connected” because every output depends on every input.
- Also called “affine” or “inner product”
Neural Net: Graphical Representation

2 layers

- Called “fully connected” because every output depends on every input.

- Also called “affine” or “inner product”

3 layers
Questions?
Neural Networks,
More generally

\[ x \rightarrow \text{Function} \rightarrow h^{(1)} \rightarrow \text{Function} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f \]
Neural Networks,
More generally

This can be:
- Fully connected layer
- Nonlinearity (ReLU, Tanh, Sigmoid)
- Convolution
- Pooling (Max, Avg)
- Vector normalization (L1, L2)
- Invent new ones
Neural Networks, More generally

\[ x \rightarrow \text{Function} \rightarrow h^{(1)} \rightarrow \text{Function} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f \]
Neural Networks, More generally
Neural Networks, More generally
Neural Networks, More generally

Here, $\theta$ represents whatever parameters that layer is using (e.g. for a fully connected layer $\theta^{(1)} = \{ W^{(1)}, b^{(1)} \}$).
Neural Networks,
More generally

Here, $\theta$ represents whatever parameters that layer is using (e.g. for a fully connected layer $\theta^{(1)} = \{ W^{(1)}, b^{(1)} \}$).
Neural Networks, More generally

Here, $\theta$ represents whatever parameters that layer is using (e.g. for a fully connected layer $\theta^{(1)} = \{ W^{(1)}, b^{(1)} \}$).
Neural Networks,
More generally

Here, $\theta$ represents whatever parameters that layer is using (e.g. for a fully connected layer $\theta^{(1)} = \{ W^{(1)}, b^{(1)} \}$).

Recall: the loss “L” measures how far the predictions “f” are from the labels “y”. The most common loss is Softmax.
Neural Networks, More generally
Neural Networks,
More generally

Goal: Find a value for parameters \((\theta^{(1)}, \theta^{(2)}, \ldots)\), so that the loss \((L)\) is small
Neural Networks,
More generally

\[ x \rightarrow \theta^{(1)} \rightarrow h^{(1)} \rightarrow \theta^{(2)} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow L \]

- To keep things clean, we will sometimes hide “y”, but remember that it is always there
Neural Networks, More generally

- To keep things clean, we will sometimes hide “y”, but remember that it is always there

- From last class, we learned that to improve the weights, we can take a step in the negative gradient direction:
Neural Networks,  
More generally

\[ x \rightarrow \text{Function} \rightarrow h^{(1)} \rightarrow \text{Function} \rightarrow h^{(2)} \rightarrow \ldots \rightarrow L \]

- To keep things clean, we will sometimes hide “y”, but remember that it is always there.

- From last class, we learned that to improve the weights, we can take a step in the negative gradient direction:

\[ \theta \leftarrow \theta - \alpha \frac{\partial L}{\partial \theta} \]
Neural Networks,
More generally

- To keep things clean, we will sometimes hide “y”, but remember that it is always there

- From last class, we learned that to improve the weights, we can take a step in the negative gradient direction:

\[
\theta \leftarrow \theta - \alpha \frac{\partial L}{\partial \theta}
\]

- How do we compute this?
Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California,
San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University,
Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal ‘hidden’ units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure¹.

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are ‘feature analysers’ between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, \( x_j \), to unit \( j \) is a linear function of the outputs, \( y_i \), of the units that are connected to \( j \) and of the weights, \( w_{ji} \), on these connections

\[
x_j = \sum_i y_i w_{ji}
\]

(1)
Backpropagation
Backpropagation


Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92037, USA

more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are ‘feature analysers’ between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input-output behaviour. This amounts to deciding what these
Forward Propagation:

\[ x \rightarrow \theta^{(1)} \rightarrow \text{Function} \rightarrow h^{(1)} \rightarrow \ldots \rightarrow \theta^{(n)} \rightarrow \text{Function} \rightarrow f \rightarrow L \]
Backpropagation

Forward Propagation:

\[ x \rightarrow \theta^{(1)} \rightarrow \text{Function} \rightarrow h^{(1)} \rightarrow \cdots \rightarrow \theta^{(n)} \rightarrow \text{Function} \rightarrow f \rightarrow L \]

(First layer)

(Last layer)

Backward Propagation:
Forward Propagation:

\[ x \rightarrow \theta^{(1)} \rightarrow h^{(1)} \rightarrow \cdots \rightarrow \theta^{(n)} \rightarrow f \rightarrow L \]

(First layer)

(Last layer)

Backward Propagation:
Backpropagation

Forward Propagation:

\[ x \rightarrow \underbrace{\text{Function}}_{(First\ layer)} \rightarrow h^{(1)} \rightarrow \cdots \rightarrow \underbrace{\text{Function}}_{(Last\ layer)} \rightarrow f \rightarrow L \]

Backward Propagation:

\[ \frac{\partial L}{\partial f} \leftarrow L \]
Backpropagation

Forward Propagation:

\[ x \to \theta^{(1)} \to \text{Function} \to h^{(1)} \to \ldots \to \theta^{(n)} \to \text{Function} \to f \to L \]

(First layer) (Last layer)

Backward Propagation:

\[ \frac{\partial L}{\partial \theta^{(n)}} \to \text{Function} \leftarrow \frac{\partial L}{\partial f} \leftarrow L \]
Backpropagation

Forward Propagation:

\[ x \rightarrow \theta^{(1)} \rightarrow \text{Function} \rightarrow h^{(1)} \rightarrow \ldots \rightarrow \theta^{(n)} \rightarrow \text{Function} \rightarrow f \rightarrow L \]

(First layer)

(Last layer)

Backward Propagation:

\[ \frac{\partial L}{\partial h^{(1)}} \rightarrow \ldots \rightarrow \frac{\partial L}{\partial \theta^{(n)}} \rightarrow \text{Function} \rightarrow \frac{\partial L}{\partial f} \rightarrow L \]
Backpropagation

Forward Propagation:
\[ x \rightarrow \text{Function} \rightarrow h^{(1)} \rightarrow \ldots \rightarrow \text{Function} \rightarrow f \rightarrow L \]

Backward Propagation:
\[ \frac{\partial L}{\partial \theta^{(1)}} \rightarrow \text{Function} \rightarrow \frac{\partial L}{\partial h^{(1)}} \rightarrow \ldots \rightarrow \text{Function} \rightarrow \frac{\partial L}{\partial f} \rightarrow L \]
Backpropagation

Backward Propagation:

\[ \frac{\partial L}{\partial x} \rightarrow \frac{\partial L}{\partial \theta^{(1)}} \rightarrow \text{Function} \rightarrow \frac{\partial L}{\partial \theta^{(2)}} \rightarrow \text{Function} \rightarrow \cdots \rightarrow L \]
Backpropagation

Backward Propagation:

\[
\frac{\partial L}{\partial \theta^{(1)}} \quad \frac{\partial L}{\partial \theta^{(2)}}
\]

\[
\frac{\partial L}{\partial x} \quad \frac{\partial L}{\partial h^{(1)}} \quad \frac{\partial L}{\partial h^{(2)}}
\]

First, compute this...
Backpropagation

Backward Propagation:

First, compute this $\frac{\partial L}{\partial \theta^{(1)}}$

Then compute this $\frac{\partial L}{\partial h^{(1)}}$

$\frac{\partial L}{\partial \theta^{(2)}}$

$\frac{\partial L}{\partial h^{(2)}}$

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

$\cdots \rightarrow L$
Backpropagation

Backward Propagation:

\[ \frac{\partial L}{\partial \theta^{(1)}} \]

\[ \frac{\partial L}{\partial h^{(1)}} \]

\[ \frac{\partial L}{\partial h^{(2)}} \]

Then compute this

First, compute this

Key observation: You can compute \( \frac{\partial L}{\partial \theta^{(2)}} \) and \( \frac{\partial L}{\partial h^{(1)}} \) given only \( \frac{\partial L}{\partial h^{(2)}} \), and you can do it one layer at a time.
Chain rule recap

I hope everyone remembers the chain rule:

\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial x}
\]
Chain rule recap

I hope everyone remembers the chain rule:

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

Forward propagation: \( x \rightarrow h \rightarrow \ldots \)

Backward propagation: \( \frac{\partial L}{\partial x} \leftarrow \frac{\partial L}{\partial h} \leftarrow \ldots \)
Chain rule recap

I hope everyone remembers the chain rule:

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

Forward propagation: \( x \rightarrow h \rightarrow \ldots \)

Backward propagation: \( \frac{\partial L}{\partial x} \leftarrow \frac{\partial L}{\partial h} \leftarrow \ldots \)

... but what if \( x \) and \( y \) are multi-dimensional?