Lecture 5: Neural Networks
Waitlist update

I was confused about the way waitlists work on Monday =( 

We have set enrollment sizes of 35 / 85 for 498 / 598

Each day overrides will be sent automatically in waitlist order to fill up to capacity

If you don’t enroll within a day of getting an override you will be dropped from the waitlist
Assignment 1

Was due on Sunday

If you use all 3 late days then you can turn it in today with no penalty

If you enrolled late, your A1 will be due **one week from the time you enrolled**
Assignment 2

Due Monday, September 30

Much longer than A1 – Start early

Your submission **must** pass the [validation script](#) to be graded!

We will be lenient on A1 submissions, but starting with A2 we will not grade your assignment if it does not pass the validation script
Where we are:

1. Use **Linear Models** for image classification problems

2. Use **Loss Functions** to express preferences over different choices of weights

3. Use **Stochastic Gradient Descent** to minimize our loss functions and train the model

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

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

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

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

\[ v = \theta \]

\[ \text{for } t \text{ in range(num_steps):} \]
\[ \text{dw} = \text{compute_gradient}(w) \]
\[ v = \rho * v + dw \]
\[ w = \text{learning_rate} * v \]
Problem: Linear Classifiers aren’t that powerful

Geometric Viewpoint

Visual Viewpoint
One template per class:
Can’t recognize different modes of a class
One solution: **Feature Transforms**

Original space

\[
r = (x^2 + y^2)^{1/2}
\]

\[
\theta = \tan^{-1}(y/x)
\]

Feature transform
One solution: **Feature Transforms**

Original space

Feature space

$r = (x^2 + y^2)^{1/2}$

$\theta = \tan^{-1}(y/x)$
One solution: **Feature Transforms**

\[
\begin{align*}
\text{Original space} & \\
\begin{aligned}
\theta &= \tan^{-1}(y/x) \\
r &= (x^2 + y^2)^{1/2}
\end{aligned} \\
\text{Feature space} & \\
\begin{aligned}
\theta & \\
r &
\end{aligned}
\end{align*}
\]
One solution: Feature Transforms

Original space

Feature space

Nonlinear classifier in original space!

Linear classifier in feature space

\[
\begin{align*}
    r &= (x^2 + y^2)^{1/2} \\
    \theta &= \tan^{-1}(y/x)
\end{align*}
\]
Image Features: Color Histogram

Ignores texture, spatial positions

Frog image is in the public domain
Image Features: Histogram of Oriented Gradients (HoG)

1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Image Features: Histogram of Oriented Gradients (HoG)

1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Example: 320x240 image gets divided into 40x30 bins; 8 directions per bin; feature vector has 30*40*9 = 10,800 numbers

Lowe, "Object recognition from local scale-invariant features", ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005
Image Features: Histogram of Oriented Gradients (HoG)

1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Example: 320x240 image gets divided into 40x30 bins; 8 directions per bin; feature vector has 30*40*9 = 10,800 numbers

Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005
Image Features: Histogram of Oriented Gradients (HoG)

1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Example: 320x240 image gets divided into 40x30 bins; 8 directions per bin; feature vector has 30*40*9 = 10,800 numbers

Image Features: Bag of Words (Data-Driven!)

Step 1: Build codebook

- Extract random patches

Cluster patches to form “codebook” of “visual words”

Fei-Fei and Perona, “A bayesian hierarchical model for learning natural scene categories”, CVPR 2005

Car image in CC0 1.0 public domain
Image Features: Bag of Words (Data-Driven!)

Step 1: Build codebook

- Extract random patches
- Cluster patches to form “codebook” of “visual words”

Step 2: Encode images

Fei-Fei and Perona, “A bayesian hierarchical model for learning natural scene categories”, CVPR 2005
Image Features
Example: Winner of 2011 ImageNet challenge

Low-level feature extraction \( \approx 10k \) patches per image
- SIFT: 128-dim
- color: 96-dim
\( \{ \) reduced to 64-dim with PCA\( \}

FV extraction and compression:
- \( N=1,024 \) Gaussians, \( R=4 \) regions \( \Rightarrow 520K \) dim x 2
- compression: \( G=8, b=1 \) bit per dimension

One-vs-all SVM learning with SGD

Late fusion of SIFT and color systems

Image Features

Feature Extraction

f

10 numbers giving scores for classes

training
Image Features vs Neural Networks

Feature Extraction

$f$

10 numbers giving scores for classes

training

10 numbers giving scores for classes

training

(Before) Linear score function:

$$f = Wx$$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$
Neural Networks

(Before) Linear score function:

Now 2-layer Neural Network

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

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

(In practice we will usually add a learnable bias at each layer as well)
Neural Networks

(Before) Linear score function:

\[ f = Wx \]

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

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

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

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

(In practice we will usually add a learnable bias at each layer as well)
Neural Networks

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

(Now) 2-layer Neural Network

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

Input: 3072
Hidden layer: 100
Output: 10

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

(Before) Linear score function:

$$f = Wx$$

(Now) 2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

Element $(i, j)$ of $W_1$ gives the effect on $h_i$ from $x_j$

Element $(i, j)$ of $W_2$ gives the effect on $s_i$ from $h_j$

Input: 3072

Hidden layer: 100

Output: 10

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$
Neural Networks

(Before) Linear score function:

\[ f = Wx \]

(Now) 2-layer Neural Network

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

Element \((i, j)\) of \(W_1\) gives the effect on \(h_i\) from \(x_j\)

All elements of \(x\) affect all elements of \(h\)

Fully-connected neural network

Also “Multi-Layer Perceptron” (MLP)
Neural Networks

Linear classifier: One template per class

(Before) Linear score function:

(Now) 2-layer Neural Network

Input: 3072
Hidden layer: 100
Output: 10

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

Neural net: first layer is bank of templates; Second layer recombines templates

(Before) Linear score function:

(Now) 2-layer Neural Network

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

Can use different templates to cover multiple modes of a class!

(Before) Linear score function:

(Now) 2-layer Neural Network

Input: 3072

Hidden layer: 100

Output: 10

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

“Distributed representation”: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:

Neural Networks

Distributed representation: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

Before Linear score function:
Deep Neural Networks

Depth = number of layers

Width: Size of each layer

Input: 3072

Output: 10

\[ s = W_6 \max(0, W_6 \max(0, W_5 \max(0, W_4 \max(0, W_3 \max(0, W_2 \max(0, W_1 x))))))) \]
Activation Functions

2-layer Neural Network

The function $ReLU(z) = \max(0, z)$ is called “Rectified Linear Unit”

This is called the **activation function** of the neural network

$$f = W_2 \max(0, W_1 x)$$
Activation Functions

2-layer Neural Network

The function $ReLU(z) = \max(0, z)$ is called “Rectified Linear Unit”

This is called the activation function of the neural network

Q: What happens if we build a neural network with no activation function?

$$s = W_2W_1x$$
Activation Functions

2-layer Neural Network

The function $ReLU(z) = \max(0, z)$ is called “Rectified Linear Unit”

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

This is called the activation function of the neural network.

Q: What happens if we build a neural network with no activation function?

\[ s = W_2 W_1 x \]

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

A: We end up with a linear classifier!
Activation Functions

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

\text{tanh} \space \tanh(x)

\text{ReLU} \space \max(0, x)

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

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

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

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

\[ \tanh(x) \]

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

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

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

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 Net in <20 lines!

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

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
    h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    dy_pred = 2.0 * (y_pred - y)
dw2 = h.T.dot(dy_pred)
dh = dy_pred.dot(w2.T)
dw1 = x.T.dot(dh * h * (1 - h))
w1 -= 1e-4 * dw1
w2 -= 1e-4 * dw2
```
```python
import numpy as np
from numpy.random import randn

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
    h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
    y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    dy_pred = 2.0 * (y_pred - y)
    dw2 = h.T.dot(dy_pred)
    dh = dy_pred.dot(w2.T)
    dw1 = x.T.dot(dh * h * (1 - h))
    w1 -= 1e-4 * dw1
    w2 -= 1e-4 * dw2
```

Initialize weights and data
Neural Net in <20 lines!

1. Initialize weights and data
2. Compute loss (sigmoid activation, L2 loss)

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

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)

for t in range(10000):
    h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
    y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    dy_pred = 2.0 * (y_pred - y)
    dw2 = h.T.dot(dy_pred)
    dh = dy_pred.dot(w2.T)
    dw1 = x.T.dot(dh * h * (1 - h))
    w1 -= 1e-4 * dw1
    w2 -= 1e-4 * dw2
```
Neural Net in <20 lines!

Initialize weights and data

Compute loss (sigmoid activation, L2 loss)

Compute gradients

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

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)

for t in range(10000):
    h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    dy_pred = 2.0 * (y_pred - y)
dw2 = h.T.dot(dy_pred)
dh = dy_pred.dot(w2.T)
dw1 = x.T.dot(dh * h * (1 - h))
w1 -= 1e-4 * dw1
w2 -= 1e-4 * dw2
```

Neural Net in <20 lines!

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, Din, H, Dout = 64, 1000, 100, 10
5 x, y = randn(N, Din), randn(N, Dout)
6 w1, w2 = randn(Din, H), randn(H, Dout)
7 for t in range(10000):
8     h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
9     y_pred = h.dot(w2)
10    loss = np.square(y_pred - y).sum()
11    dy_pred = 2.0 * (y_pred - y)
12    dw2 = h.T.dot(dy_pred)
13    dh = dy_pred.dot(w2.T)
14    dw1 = x.T.dot(dh * h * (1 - h))
15    w1 -= 1e-4 * dw1
16    w2 -= 1e-4 * dw2
```
Our brains are made of Neurons

![Neuron diagram](image)

*Neuron image by Felipe Perucho is licensed under [CC-BY 3.0](https://creativecommons.org/licenses/by/3.0)*
Our brains are made of Neurons

- Synapse
- Cell body
- Dendrite
- Axon
- Presynaptic terminal
Our brains are made of Neurons

- **Cell body**
- **Dendrite**
- **Axon**
- **Presynaptic terminal**

Impulses carried toward cell body

Impulses carried away from cell body
Our brains are made of Neurons

 ![Diagram of a neuron showing key components: Cell body, Dendrite, Axon, Synapse, Presynaptic terminal, Impulses carried toward cell body, Impulses carried away from cell body, and a graph showing firing rate as a nonlinear function of inputs.]
Biological Neuron

dendrite

axon

cell body

presynaptic terminal

Artificial Neuron

input layer

hidden layer 1

hidden layer 2

output layer

$x_0$

$w_0$

synapse

axon from a neuron

dendrite

$w_0 x_0$

$w_1 x_1$

$w_2 x_2$

$\sum w_i x_i + b$

$\sum_i w_i x_i + b$

activation function

$f$

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

Justin Johnson

Lecture 5 - 48

September 18, 2019
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”, ICCV 2019
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
- Rate code may not be adequate

[Dendritic Computation. London and Hausser]
Space Warping

Consider a linear transform: $h = Wx$
Where $x$, $h$ are both 2-dimensional
Consider a linear transform: $h = Wx$
Where $x$, $h$ are both 2-dimensional
Space Warping

Consider a linear transform: \( h = Wx \)
Where \( x \), \( h \) are both 2-dimensional
Points not linearly separable in original space

Consider a linear transform: $h = Wx$
Where $x$, $h$ are both 2-dimensional
Space Warping

Points not linearly separable in original space

Consider a linear transform: $h = Wx$
Where $x$, $h$ are both 2-dimensional

Feature transform: $h = Wx$

Not linearly separable in feature space
Space Warping

Consider a neural net hidden layer:
\[ h = \text{ReLU}(Wx) = \max(0, Wx) \]
Where \( x, h \) are both 2-dimensional

Feature transform:
\[ h = \text{ReLU}(Wx) \]
Consider a neural net hidden layer: 
\[ h = \text{ReLU}(Wx) = \max(0, Wx) \]
Where \( x, h \) are both 2-dimensional.
Consider a neural net hidden layer:
\[ h = \text{ReLU}(Wx) = \max(0, Wx) \]
Where \( x, h \) are both 2-dimensional.

**Space Warping**

Feature transform:
\[ h = \text{ReLU}(Wx) \]

B is “collapsed” onto +h2 axis

---

Justin Johnson  
Lecture 5 - 59  
September 18, 2019
Consider a neural net hidden layer:
\[ h = \text{ReLU}(Wx) = \max(0, Wx) \]
Where \( x, h \) are both 2-dimensional

Feature transform:
\[ h = \text{ReLU}(Wx) \]

B is “collapsed” onto +h2 axis
D “collapsed” onto +h1 axis
Consider a neural net hidden layer: $h = \text{ReLU}(Wx) = \max(0, Wx)$

Where $x$, $h$ are both 2-dimensional.

Feature transform:

- $h = \text{ReLU}(Wx)$
- $B$ is “collapsed” onto $+h_2$ axis
- $C$ “collapsed” onto origin
- $D$ “collapsed” onto $+h_1$ axis
Consider a neural net hidden layer:

\[ h = \text{ReLU}(Wx) = \max(0, Wx) \]

Where \( x, h \) are both 2-dimensional

Feature transform:

\[ h = Wx \]

Points not linearly separable in original space.
Consider a neural net hidden layer: 
\[ h = \text{ReLU}(Wx) = \max(0, Wx) \]
Where \( x, h \) are both 2-dimensional

Points not linearly separable in original space

Feature transform:
\[ h = \text{ReLU}(Wx) \]
Consider a neural net hidden layer:

\[ h = \text{ReLU}(Wx) = \max(0, Wx) \]

Where \( x, h \) are both 2-dimensional

Points not linearly separable in original space

Feature transform:

\[ h = \text{ReLU}(Wx) \]

Points are linearly separable in features space!
Consider a neural net hidden layer:

\[ h = \text{ReLU}(Wx) = \max(0, Wx) \]

Where \( x, h \) are both 2-dimensional

**Feature transform:**

\[ h = \text{ReLU}(Wx) \]

Points are linearly separable in features space!
Setting the number of layers and their sizes

More hidden units = more capacity
Don’t regularize with size; instead use stronger L2

\[
\lambda = 0.001 \quad \lambda = 0.01 \quad \lambda = 0.1
\]

(Web demo with ConvNetJS: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html)
Universal Approximation

A neural network with one hidden layer can approximate any function $f: \mathbb{R}^N \rightarrow \mathbb{R}^M$ with arbitrary precision.

*Many technical conditions: Only holds on compact subsets of $\mathbb{R}^n$; function must be continuous; need to define “arbitrary precision”; etc.
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network

\[ x \rightarrow w_{1}h_{1} + u_{1} \rightarrow h_{2} \rightarrow w_{2}h_{2} + u_{2} \rightarrow h_{3} \rightarrow w_{3}h_{3} + u_{3} \rightarrow y \]

Input: $x (1,)$

First layer weights: $w (3,1)$
First layer bias: $b (3,)$

Second layer weights: $u (1,3)$
First layer bias: $p (1,)$

Output: $y (1,)$
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network

Input: $x \ (1,)$

First layer weights: $w \ (3,1)$
First layer bias: $b \ (3,)$

Second layer weights: $u \ (1,3)$
First layer bias: $p \ (1,)$

$h_1 = \max(0, w_1 \ast x + b_1)$
$h_2 = \max(0, w_2 \ast x + b_2)$
$h_3 = \max(0, w_3 \ast x + b_3)$
$y = u_1 \ast h_1 + u_2 \ast h_2 + u_3 \ast h_3 + p$
Universal Approximation

Example: Approximating a function \( f: \mathbb{R} \rightarrow \mathbb{R} \) with a two-layer ReLU network

\[
\begin{align*}
  h_1 &= \text{max}(0, w_1 \times x + b_1) \\
  h_2 &= \text{max}(0, w_2 \times x + b_2) \\
  h_3 &= \text{max}(0, w_3 \times x + b_3) \\
  y &= u_1 \times h_1 + u_2 \times h_2 + u_3 \times h_3 + p
\end{align*}
\]
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network

- Input: $x \ (1,)$
- First layer weights: $w \ (3,1)$
- First layer bias: $b \ (3,)$
- Second layer weights: $u \ (1,3)$
- First layer bias: $p \ (1,)$

$h_1 = \max(0, w_1 \cdot x + b_1)$
$h_2 = \max(0, w_2 \cdot x + b_2)$
$h_3 = \max(0, w_3 \cdot x + b_3)$

$y = u_1 \cdot h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p$

Output is a sum of shifted, scaled ReLUs:
Universal Approximation

Example: Approximating a function \( f: \mathbb{R} \rightarrow \mathbb{R} \) with a two-layer ReLU network

- **Input:** \( x \) \((1,)
- **First layer weights:** \( w \) \((3,1)\)
- **First layer bias:** \( b \) \((3,)\)
- **Second layer weights:** \( u \) \((1,3)\)
- **First layer bias:** \( p \) \((1,)\)

**Output:** \( y \) \((1,)\)

- **Output is a sum of shifted, scaled ReLUs:**

\[
\begin{align*}
\text{h1} &= \max(0, w_1 \ast x + b_1) \\
\text{h2} &= \max(0, w_2 \ast x + b_2) \\
\text{h3} &= \max(0, w_3 \ast x + b_3) \\
y &= u_1 \ast \text{h1} + u_2 \ast \text{h2} + u_3 \ast \text{h3} + p
\end{align*}
\]
Universal Approximation

Example: Approximating a function \( f: \mathbb{R} \rightarrow \mathbb{R} \) with a two-layer ReLU network

\[
\begin{align*}
    h_1 &= \max(0, w_1 \cdot x + b_1) \\
    h_2 &= \max(0, w_2 \cdot x + b_2) \\
    h_3 &= \max(0, w_3 \cdot x + b_3) \\
    y &= u_1 \cdot h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p
\end{align*}
\]

We can build a “bump function” using four hidden units.
Universal Approximation

Example: Approximating a function \( f: \mathbb{R} \rightarrow \mathbb{R} \) with a two-layer ReLU network

\[
\begin{align*}
\text{Input:} & \quad x \ (1,) \\
\text{First layer weights:} & \quad w \ (3,1) \\
\text{First layer bias:} & \quad b \ (3,) \\
\text{Second layer weights:} & \quad u \ (1,3) \\
\text{First layer bias:} & \quad p \ (1,)
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\text{h}_1 &= \max(0, w_1 \ast x + b_1) \\
\text{h}_2 &= \max(0, w_2 \ast x + b_2) \\
\text{h}_3 &= \max(0, w_3 \ast x + b_3) \\
y &= u_1 \ast \text{h}_1 + u_2 \ast \text{h}_2 + u_3 \ast \text{h}_3 + p \\
\end{array}
\end{align*}
\]

We can build a “bump function” using four hidden units:

\[
\begin{align*}
m_1 &= t / (s_2 - s_1) \\
m_2 &= t / (s_4 - s_3)
\end{align*}
\]
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network

\[ h_1 = \max(0, w_1 \cdot x + b_1) \]
\[ h_2 = \max(0, w_2 \cdot x + b_2) \]
\[ h_3 = \max(0, w_3 \cdot x + b_3) \]
\[ y = u_1 \cdot h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p \]

We can build a “bump function” using four hidden units

\[ m_1 = \frac{t}{s_2 - s_1} \]
\[ m_2 = \frac{t}{s_4 - s_3} \]

\[ m_1 = \max(0, x - s_1) \]
\[ m_2 = \max(0, x - s_3) \]
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network

Input: $x (1,)$
First layer weights: $w$ (3,1)
First layer bias: $b$ (3,)
Second layer weights: $u$ (1,3)
First layer bias: $p$ (1,)
Output: $y (1,)$

$h_1 = \max(0, w_1 \cdot x + b_1)$
$h_2 = \max(0, w_2 \cdot x + b_2)$
$h_3 = \max(0, w_3 \cdot x + b_3)$
$y = u_1 \cdot h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p$

$y = u_1 \cdot \max(0, w_1 \cdot x + b_1) + u_2 \cdot \max(0, w_2 \cdot x + b_2) + u_3 \cdot \max(0, w_3 \cdot x + b_3) + p$

We can build a “bump function” using four hidden units

$m_1 = t / (s_2 - s_1)$
$m_2 = t / (s_4 - s_3)$

$m_1 \cdot \max(0, x - s_1)$
$m_1 \cdot \max(0, x - s_2)$
$-m_1 \cdot \max(0, x - s_2)$

$m_2 \cdot (x - s_3)$
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network

Input: $x \ (1,)$
First layer weights: $w \ (3,1)$
First layer bias: $b \ (3,)$
Second layer weights: $u \ (1,3)$
First layer bias: $p \ (1,)$

Output: $y \ (1,)$

$h_1 = \max(0, w_1 \cdot x + b_1)$
$h_2 = \max(0, w_2 \cdot x + b_2)$
$h_3 = \max(0, w_3 \cdot x + b_3)$

$y = u_1 \cdot h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p$

We can build a “bump function” using four hidden units

$m_1 = \frac{t}{s_2 - s_1}$
$m_2 = \frac{t}{s_4 - s_3}$

$m_1 \cdot \max(0, x - s_1)$
$-m_1 \cdot \max(0, x - s_2)$
$-m_2 \cdot \max(0, x - s_3)$
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network

Input: $x \ (1,)$  
First layer weights: $w \ (3,1)$  
First layer bias: $b \ (3,)$  
Output: $y \ (1,)$  
Second layer weights: $u \ (1,3)$  
First layer bias: $p \ (1,)$

$h_1 = \max(0, w_1 \cdot x + b_1)$
$h_2 = \max(0, w_2 \cdot x + b_2)$
$h_3 = \max(0, w_3 \cdot x + b_3)$
$y = u_1 \cdot h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p$

We can build a “bump function” using four hidden units:

$m_1 = t / (s_2 - s_1)$
$m_2 = t / (s_4 - s_3)$

$m_1 \cdot \max(0, x - s_1) - m_1 \cdot \max(0, x - s_2)$
$m_2 \cdot \max(0, x - s_3) - m_2 \cdot \max(0, x - s_4)$
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network

Input: $x \ (1,)$

First layer weights: $w \ (3,1)$
First layer bias: $b \ (3,)$

Second layer weights: $u \ (1,3)$
First layer bias: $p \ (1,)$

Output: $y \ (1,)$

$$h_1 = \text{max}(0, w_1 \times x + b_1)$$
$$h_2 = \text{max}(0, w_2 \times x + b_2)$$
$$h_3 = \text{max}(0, w_3 \times x + b_3)$$
$$y = u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p$$

We can build a “bump function” using four hidden units

With 4K hidden units we can build a sum of $K$ bumps
Universal Approximation

Example: Approximating a function \( f: \mathbb{R} \rightarrow \mathbb{R} \) with a two-layer ReLU network

\[
\begin{align*}
\text{Input: } & \quad x \ (1,) \\
\text{First layer weights: } & \quad w \ (3,1) \\
\text{First layer bias: } & \quad b \ (3,) \\
\text{Second layer weights: } & \quad u \ (1,3) \\
\text{First layer bias: } & \quad p \ (1,)
\end{align*}
\]

\[
\begin{align*}
h_1 &= \max(0, w_1 \cdot x + b_1) \\
h_2 &= \max(0, w_2 \cdot x + b_2) \\
h_3 &= \max(0, w_3 \cdot x + b_3) \\
y &= u_1 \cdot h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p
\end{align*}
\]

We can build a “bump function” using four hidden units.

With 4K hidden units we can build a sum of K bumps.

Approximate functions with bumps!
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network

$$\begin{align*}
\text{Input: } & \quad x \ (1,) \\
\text{First layer weights: } & \quad w \ (3,1) \\
\text{First layer bias: } & \quad b \ (3,) \\
\text{Second layer weights: } & \quad u \ (1,3) \\
\text{First layer bias: } & \quad p \ (1,) \\
\text{Output: } & \quad y \ (1,) \\

h_1 &= \max(0, w_1 \cdot x + b_1) \\
h_2 &= \max(0, w_2 \cdot x + b_2) \\
h_3 &= \max(0, w_3 \cdot x + b_3) \\
y &= u_1 \cdot \max(0, w_1 \cdot x + b_1) + u_2 \cdot \max(0, w_2 \cdot x + b_2) + u_3 \cdot \max(0, w_3 \cdot x + b_3) + p
\end{align*}$$

What about...
- Gaps between bumps?
- Other nonlinearities?
- Higher-dimensional functions?

See Nielsen, Chapter 4

Approximate functions with bumps!
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_3 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1) + u_2 \max(0, w_2 x + b_2) + u_3 \max(0, w_3 x + b_3) + p$$

Reality check: Networks don’t really learn bumps!
Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network

$$x \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow y$$

Input: $x \ (1,)$

Output: $y \ (1,)$

Universal approximation tells us:
- Neural nets can represent any function

Universal approximation DOES NOT tell us:
- Whether we can actually learn any function with SGD
- How much data we need to learn a function

Remember: kNN is also a universal approximator!
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1],$

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$
Convex Functions

A function \( f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R} \) is **convex** if for all \( x_1, x_2 \in X, t \in [0, 1], \)

\[
f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)
\]

Example: \( f(x) = x^2 \) is convex:
Convex Functions

A function \( f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R} \) is \textit{convex} if for all \( x_1, x_2 \in X, t \in [0, 1], \)

\[
f(t x_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)
\]

Example: \( f(x) = x^2 \) is convex:
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1],$

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

Example: $f(x) = x^2$ is convex:

![Graph showing convex function]
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

Example: $f(x) = \cos(x)$ is **not** convex:
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is convex if for all $x_1, x_2 \in X, t \in [0, 1]$, 

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

*Many technical details! See e.g. IOE 661 / MATH 663*
Convex Functions

A function \( f: X \subseteq \mathbb{R}^N \rightarrow \mathbb{R} \) is **convex** if for all \( x_1, x_2 \in X, t \in [0, 1], \)

\[
f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)
\]

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize:** can derive theoretical guarantees about
converging to global minimum*

*Many technical details! See e.g. IOE 661 / MATH 663
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1],$

$$ f(t x_1 + (1 - t) x_2) \leq t f(x_1) + (1 - t) f(x_2) $$

**Intuition**: A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about converging to global minimum

Linear classifiers optimize a convex function!

- $s = f(x; W) = W x$
- $L_i = - \log \left( \frac{e^{s y_i}}{\sum_j e^{s_j}} \right)$ **Softmax**
- $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$ **SVM**
- $L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$

R(W) = L2 or L1 regularization

*Many technical details! See e.g. IOE 661 / MATH 663*
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is convex if for all $x_1, x_2 \in X, t \in [0, 1],$

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are easy to optimize: can derive theoretical guarantees about converging to global minimum*

*Many technical details! See e.g. IOE 661 / MATH 663

Neural net losses sometimes look convex-ish:

1D slice of loss landscape for a 4-layer ReLU network with 10 input features, 32 units per hidden layer, 10 categories, with softmax loss
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize:** can derive theoretical guarantees about converging to global minimum*

*Many technical details! See e.g. IOE 661 / MATH 663
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is \textbf{convex} if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

\textbf{Intuition:} A convex function is a (multidimensional) bowl.

Generally speaking, convex functions are \textbf{easy to optimize}: can derive theoretical guarantees about converging to global minimum*

\*Many technical details! See e.g. IOE 661 / MATH 663
Convex Functions

A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is convex if for all $x_1, x_2 \in X, t \in [0, 1]$,
$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl.

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about converging to global minimum.

*Many technical details! See e.g. IOE 661 / MATH 663*
Convex Functions

A function \( f : X \subseteq \mathbb{R}^N \to \mathbb{R} \) is **convex** if for all \( x_1, x_2 \in X, t \in [0, 1], \)

\[
f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)
\]

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize:** can derive theoretical guarantees about converging to global minimum*

Most neural networks need **nonconvex optimization**
- Few or no guarantees about convergence
- Empirically it seems to work anyway
- Active area of research

*Many technical details! See e.g. IOE 661 / MATH 663
Summary

Feature transform + Linear classifier allows nonlinear decision boundaries

Original space

Feature space

Nonlinear classifier in original space!

Feature Extraction

10 numbers giving scores for classes

Neural Networks as learnable feature transforms

10 numbers giving scores for classes
Summary

From linear classifiers to fully-connected networks

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

Input: 3072
Hidden layer: 100
Output: 10

Linear classifier: One template per class
Neural networks: Many reusable templates
Summary

From linear classifiers to fully-connected networks

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

Neural networks loosely inspired by biological neurons but be careful with analogies
Summary

From linear classifiers to fully-connected networks

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

- Input: 3072
- Hidden layer: 100
- Output: 10

- Space Warping
- Universal Approximation
- Nonconvex
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 \( \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2} \) n we can learn \( W_1 \) and \( W_2 \)
Next time: Backpropagation