Lecture 5: Neural Networks
Assignment 2

• Use SGD to train linear classifiers and fully-connected networks
• After today, can do full assignment
• If you have a hard time computing derivatives, wait for next Monday’s lecture on backprop
• Due Friday September 25, 11:59pm EDT
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 **Regularization** to prevent overfitting to training data
4. Use **Stochastic Gradient Descent** to minimize our loss functions and train the model

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

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

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

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

\[ v = 0 \]
for \( t \) in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w = 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

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

Feature space

**Feature transform**
One solution: Feature Transforms

Original space

Feature space

Feature transform

Linear classifier in feature space

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

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

Original space

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

Feature transform

Feature space

Nonlinear classifier in original space!

Linear classifier in feature space!
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

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

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: 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: 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 ≈ 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 ⇒ 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

10 numbers giving scores for classes

10 numbers giving scores for classes

Neural Networks

**Input:** $x \in \mathbb{R}^D$ \hspace{1cm} **Output:** $f(x) \in \mathbb{R}^C$

**Before:** Linear Classifier: \hspace{1em} $f(x) = Wx + b$

Learnable parameters: \hspace{1em} $W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C$
Neural Networks

**Input:** \( x \in \mathbb{R}^D \) \hspace{1cm} **Output:** \( f(x) \in \mathbb{R}^C \)

**Before:** Linear Classifier: \( f(x) = Wx + b \)
Learnable parameters: \( W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C \)

**Now:** Two-Layer Neural Network: \( f(x) = W_2 \max(0, W_1x + b_1) + b_2 \)
Neural Networks

Input: $x \in \mathbb{R}^D$  
Output: $f(x) \in \mathbb{R}^C$

Before: Linear Classifier:  
$f(x) = Wx + b$
Learnable parameters: $W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C$

Now: Two-Layer Neural Network:  
$f(x) = W_2 \max(0, W_1x + b_1) + b_2$
Learnable parameters: $W_1 \in \mathbb{R}^{H \times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C \times H}, b_2 \in \mathbb{R}^C$
Neural Networks

**Input:** $x \in \mathbb{R}^D$  \hspace{1cm} **Output:** $f(x) \in \mathbb{R}^C$

**Before:** Linear Classifier: $f(x) = Wx + b$
Learnable parameters: $W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C$

**Now:** Two-Layer Neural Network: $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$
Learnable parameters: $W_1 \in \mathbb{R}^{H \times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C \times H}, b_2 \in \mathbb{R}^C$

Or Three-Layer Neural Network:
$f(x) = W_3 \max(0, W_2 \max(0, W_1 x + b_1) + b_2) + b_3$
Neural Networks

**Before**: Linear classifier

\[ f(x) = Wx + b \]

**Now**: 2-layer Neural Network

\[ f(x) = W_2 \max(0, W_1 x + b_1) + b_2 \]

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 classifier \[ f(x) = Wx + b \]

**Now:** 2-layer Neural Network \[ f(x) = W_2 \max(0, W_1 x + b_1) + b_2 \]

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

Input: 3072

Hidden layer: 100

Output: 10

Before: Linear classifier \[ f(x) = Wx + b \]

Now: 2-layer Neural Network \[ f(x) = W_2 \max(0, W_1 x + b_1) + b_2 \]

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

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

**Before**: Linear classifier  \[ f(x) = Wx + b \]

**Now**: 2-layer Neural Network  \[ f(x) = W_2 \max(0, W_1 x + b_1) + b_2 \]

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: $10$

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

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

“Distributed representation”: Most templates not interpretable!

(Before) Linear score function:

(Now) 2-layer Neural Network

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

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

$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$
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?

$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$
Activation Functions

2-layer Neural Network

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

$$f(x) = W_2 \left[ \max(0, W_1 x + b_1) + b_2 \right]$$

This is called the activation function of the neural network

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

$$f(x) = W_2 (W_1 x + b_1) + b_2$$

$$= (W_1 W_2) x + (W_2 b_1 + b_2)$$

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

**Sigmoid**

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

**tanh**

\[ \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \]

**ReLU**

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

**Leaky ReLU**

\[ \text{max}(0.2x, x) \]

**Softplus**

\[ \log(1 + \exp(x)) \]

**ELU**

\[ f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \leq 0 \end{cases} \]
Activation Functions

**Sigmoid**

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

\[ \tan x = \frac{e^{2x} - 1}{e^{2x} + 1} \]

**ReLU**

\[ \max(0, x) \]

ReLU is a good default choice for most problems

**Sigmoid**

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

**tanh**

\[ \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \]

**ReLU**

\[ \max(0, x) \]

Leaky ReLU

\[ \max(0.2x, x) \]

**Softplus**

\[ \log(1 + \exp(x)) \]

**ELU**

\[ f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \leq 0 \end{cases} \]
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
```
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
```

Initialize weights and data
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
```
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
```
Neural Net in <20 lines!

```
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
```
Our brains are made of Neurons

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

*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

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

- **Dendrite**
- **Cell body**
- **Axon**
- **Synapse**

**Impulses carried toward cell body**

**Impulses carried away from cell body**

**Presynaptic terminal**
Our brains are made of Neurons

- Cell body
- Axon
- Dendrite
- Synapse
- Presynaptic terminal

Impulses carried toward cell body
Impulses carried away from cell body

Firing rate is a nonlinear function of inputs
Biological Neuron

dendrite

cell body

presynaptic terminal

axon

Artificial Neuron

input layer

hidden layer 1

hidden layer 2

output layer

Neuron image by Felipe Perucho
is licensed under CC-BY 3.0
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
Be very careful with 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
- Rate code may not be adequate

[Dendritic Computation. London and Hausser]
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

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

Feature transform: 
\( h = Wx \)
Space Warping

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

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

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

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

B is “collapsed” onto +h2 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
- **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.
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:
\[ h = Wx \]
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.

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.
Setting the number of layers and their sizes

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

\[ \lambda = 0.001 \]

\[ \lambda = 0.01 \]

\[ \lambda = 0.1 \]

(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

```
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,)
```

![Diagram](image_url)
Universal Approximation

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

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

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

\[
\begin{align*}
  h_1 &= \max(0, w_1 \cdot x + b_1) \\
  h_2 &= \max(0, w_2 \cdot x + b_2) \\
  h_3 &= \max(0, w_3 \cdot x + b_3) \\
  y &= u_1 \cdot h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p
\end{align*}
\]
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) \quad \text{First layer bias:} \quad b \ (3,) \\
\text{Second layer weights:} & \quad u \ (1,3) \quad \text{First layer bias:} \quad p \ (1,) \\

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
\end{align*}
\]
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: $y \ (1,)$

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

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

Output is a sum of shifted, scaled ReLUs:

Flip left / right based on sign of $w_i$

Slope is given by $u_i \ast w_i$

Position of “bend” given by $b_i$
Universal Approximation

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

\[ x \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow y \]

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: $y \ (1,)$

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

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

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

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

Example: Approximating a function f: R -> 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:

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

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

$$x \xrightarrow{w_1} h_1 \xrightarrow{u_1} y$$

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

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

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

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

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

\[
\begin{align*}
m_1 &= \frac{t}{s_2 - s_1} \\
m_2 &= \frac{t}{s_4 - s_3}
\end{align*}
\]

\[
\begin{align*}
m_1 \cdot \max(0, x - s_1) - m_1 \cdot \max(0, x - s_2)
\end{align*}
\]
Universal Approximation

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

Input: $x \in \mathbb{R}$

First layer weights: $w \in \mathbb{R}^{3 \times 1}$
First layer bias: $b \in \mathbb{R}^{3 \times 1}$

Second layer weights: $u \in \mathbb{R}^{1 \times 3}$
First layer bias: $p \in \mathbb{R}^{1 \times 1}$

Output:

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

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

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

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

Output:

$y = \max(0, w_1 \cdot x + b_1) + \max(0, w_2 \cdot x + b_2) + \max(0, w_3 \cdot x + b_3) + p$
Universal Approximation

Example: Approximating a function f: R -> R with a two-layer ReLU network

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

First layer:
- h1 = max(0, w1 * x + b1)
- h2 = max(0, w2 * x + b2)
- h3 = max(0, w3 * x + b3)

Output:
y = u1 * h1 + u2 * h2 + u3 * h3 + p

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

m1 = t / (s2 – s1)
m2 = t / (s4 – s3)

y = u1 * max(0, w1 * x + b1)
+ u2 * max(0, w2 * x + b2)
+ u3 * max(0, w3 * x + b3)
+ p

m1 * max(0, x – s1)
-m1 * max(0, x – s2)
-m2 * max(0, x – s3)
m2 * max(0, x – s4)
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,)

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

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

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

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

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

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

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 \( f: \mathbb{R} \rightarrow \mathbb{R} \) 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,) \\
\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 h_1 + u_2 \cdot h_2 + u_3 \cdot h_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 $f: \mathbb{R} \to \mathbb{R}$ 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 h_1 + u_2 \cdot h_2 + u_3 \cdot h_3 + p
\end{align*}
\]

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

\[
\begin{align*}
\text{Input:} & \quad x (1,) \\
\text{Output:} & \quad y (1,)
\end{align*}
\]

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 \textbf{convex} if for all \( x_1, x_2 \in X, t \in [0, 1], \)
\[
f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)
\]
Convex Functions

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

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

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:
Convex Functions

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

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

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

![Graph showing convex function example](chart.png)
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 \( f : X \subseteq \mathbb{R}^N \to \mathbb{R} \) 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.

*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 \( x_1, x_2 \in X, t \in [0, 1] \),
\[
f(t x_1 + (1 - t) x_2) \leq t f(x_1) + (1 - t) f(x_2)
\]

**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*.

*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 $x_1, x_2 \in X$, $t \in [0, 1],$

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

**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**!

- $s = f(x; W) = Wx$
- $L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$ **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 $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ 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

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

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

**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 \( f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R} \) 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*

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

With local minima:

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 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

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

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

Feature space

Nonlinear classifier in original space!

Feature transform

Linear classifier in feature space

Neural Networks as learnable feature transforms

Feature Extraction

10 numbers giving scores for classes

training

Figure copyright Krizhevsky, Sutskever, and Hinton, 2012. Reused with permission.

10 numbers giving scores for classes

training
Summary

From linear classifiers to fully-connected networks

\[ f(x) = W_2 \max(0, W_1 x + b_1) + b_2 \]

Linear classifier: One template per class

Neural networks: Many reusable templates
Summary

From linear classifiers to fully-connected networks

\[ f(x) = W_2 \max(0, W_1 x + b_1) + b_2 \]

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

From linear classifiers to fully-connected networks

\[ f(x) = W_2 \max(0, W_1 x + b_1) + b_2 \]

Space Warping

Universal Approximation

Nonconvex

Input: 3072

Hidden layer: 100

Output: 10
Problem: How to compute gradients?

\[ s = W_2 \max(0, W_1 x + b_1) + b_2 \]  
Nonlinear score function

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

\[ R(W) = \sum_k W_k^2 \]  
L2 Regularization

\[ L(W_1, W_2, b_1, b_2) = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2) \]  
Total loss

If we can compute \( \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial b_2} \) then we can optimize with SGD
Next time:
Backpropagation