Lecture 35: Optimization and Neural Nets

CS 4670/5670
Sean Bell

Aside: “CNN” vs “ConvNet”

Note:

• There are many papers that use either phrase, but

• “ConvNet” is the preferred term, since “CNN” clashes with that other thing called CNN
Q2: At initialization, $W$ is small and thus $s \approx 0$. What is the loss $L$?
Softmax vs SVM Loss
Softmax vs SVM Loss

Softmax:

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

Softmax:

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

SVM:

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

**Softmax:**

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

assume scores:

[10, -2, 3]

[10, 9, 9]

[10, -100, -100]

and \( y_i = 0 \)

**SVM:**

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

**Softmax:**

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

**SVM:**

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

**Assume scores:**

\[
\begin{align*}
[10, -2, 3] \\
[10, 9, 9] \\
[10, -100, -100] \\
\text{and } y_i = 0
\end{align*}
\]

**Q:** Suppose I take a datapoint and I jiggle a bit (changing its score slightly). What happens to the loss in both cases?
Softmax Classifier

Let’s code this up in NumPy:

```python
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```
Softmax Classifier

Let's code this up in NumPy:

```python
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn't work — what's the problem?

\[
p_{i,j} = \frac{e^{s_{i,j}}}{\sum_{k} e^{s_{i,k}}}
\]
Let’s code this up in NumPy:

```python
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

**Doesn’t work** — what’s the problem?

- What if there is the value 1000 appears in “s”?
Let’s code this up in NumPy:

```python
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn’t work — what’s the problem?
- What if there is the value 1000 appears in “s”?

*Overflow* —> we get *inf/inf* = *NaN*
Let’s code this up in NumPy:

```python
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn’t work — what’s the problem?
- What if there is the value 1000 appears in “s”?
  
  *Overflow* —> we get inf/inf = NaN

- What if the largest value is -1000?
Softmax Classifier

Let’s code this up in NumPy:

```python
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn’t work — what’s the problem?
- What if there is the value 1000 appears in “s”?
  
  \[ p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}} \]

  \textit{Overflow} —\> we get inf/inf = NaN

- What if the largest value is -1000?
  
  \textit{Underflow} —\> we get 0/0 = NaN
Let's code this up in NumPy:

```python
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

**Doesn't work** — what's the problem?

- What if there is the value 1000 appears in “s”?
  
  *Overflow* —> we get inf/inf = NaN

- What if the largest value is -1000?
  
  *Underflow* —> we get 0/0 = NaN

This expression is numerically unstable
Softmax Classifier
Observation: subtracting a constant does not change “p”: 

**Softmax Classifier**
Observation: subtracting a constant does not change “p”:

\[ p_{i,j} = \frac{e^{s_{i,j} - C}}{\sum_k e^{s_{j,k} - C}} = \frac{e^{-C} e^{s_{i,j}}}{\sum_k e^{-C} e^{s_{i,k}}} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}} \]
Softmax Classifier

Observation: subtracting a constant does not change “p”:

\[
p_{i,j} = \frac{e^{s_{i,j} - C}}{\sum_k e^{s_{j,k} - C}} = \frac{e^{-C} e^{s_{i,j}}}{\sum_k e^{-C} e^{s_{i,k}}} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}
\]

If we choose “C” to be the max, then it works:
Softmax Classifier

Observation: subtracting a constant does not change “p”:

\[ p_{i,j} = \frac{e^{s_{i,j} - C}}{\sum_k e^{s_{j,k} - C}} = \frac{e^{-C} e^{s_{i,j}}}{\sum_k e^{-C} e^{s_{i,k}}} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}} \]

If we choose “C” to be the max, then it works:

- If a large value appears in “s”, then that value will become 1 and all others will be 0 (avoiding overflow)
Softmax Classifier

Observation: subtracting a constant does not change “p”:

\[
p_{i,j} = \frac{e^{s_{i,j} - C}}{\sum_k e^{s_{j,k} - C}} = \frac{e^{-C} e^{s_{i,j}}}{\sum_k e^{-C} e^{s_{i,k}}} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}
\]

If we choose “C” to be the max, then it works:

- If a large value appears in “s”, then that value will become 1 and all others will be 0 (avoiding overflow)
- If all values in “s” are large negative, then they will be shifted up towards 0 (avoiding underflow)
Optimization

How do we minimize the loss $L$?
Idea #1: Random Search
Idea #1: Random Search
Idea #1: Random Search
Idea #1: Random Search
Idea #2: Follow the Slope
Idea #2: Follow the Slope
Idea #2: Follow the Slope
Idea #2: Follow the Slope
Idea #2: Follow the Slope

In 1D, the derivative of a function is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

In multiple dimensions, the *gradient* is the vector of (partial derivatives).
Idea #2: Follow the Slope

In 1D, the derivative of a function is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

In multiple dimensions, the gradient is the vector of (partial derivatives).

We could directly estimate it:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$
Idea #2: Follow the Slope

In 1D, the derivative of a function is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

In multiple dimensions, the gradient is the vector of (partial derivatives).

We could directly estimate it:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

But this is super slow.
Idea #2: Follow the Slope

We can just directly compute the gradient:

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

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

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

want $\nabla_W L$
Idea #2: Follow the Slope

We can just directly compute the gradient:

\[
L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2 \\
L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\
s = f(x; W) = Wx \\
\text{want } \nabla_W L
\]
Idea #2: Follow the Slope

In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone
Idea #2: Follow the Slope

In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a gradient check.
Gradient Descent
# Vanilla Gradient Descent

```python
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
Gradient Descent

### Vanilla Gradient Descent

```python
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * weights_grad  # perform parameter update
```

- **W_1**
- **W_2**

original W

negative gradient direction
Gradient Descent

```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * weights_grad  # perform parameter update
```

Step size called the "learning rate"

negative gradient direction

original W

Slide from Karpathy 2016
Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

```python
# Vanilla Minibatch Gradient Descent

while True:
    data_batch = sample_training_data(data, 256)  # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad  # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples

- e.g. Krizhevsky ILSVRC ConvNet used 256 examples
Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

```python
# Vanilla Minibatch Gradient Descent

while True:
    data_batch = sample_training_data(data, 256)  # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad  # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

**Note:** In practice we use fancier update rules (momentum, RMSprop, ADAM, etc)
Mini-batch Gradient Descent

Example of optimization progress while training a neural network.

(Typical loss)

(Loss over mini-batches goes down over time.)
Setting the learning rate

Learning rate: $1e6$ — what could go wrong?

![Graph showing loss as a function of a weight in W]
Setting the learning rate

Learning rate: $1e6$ — what could go wrong?

![Graph showing the loss function with a weight in W](image)
Setting the learning rate

Learning rate: $1e6$ — what could go wrong?

![Diagram showing loss function with points indicating a weight in $W$.]
Setting the learning rate

Learning rate: 1e6 — what could go wrong?

\[ L \]

Loss

A weight in \( W \)
Setting the learning rate

Learning rate: 1e6 — what could go wrong?

![Diagram showing a function with a learning rate of 1e6](image)

$L$ represents the loss, and $A$ weight in $W$.
Setting the learning rate

Learning rate: $1e6$ — what could go wrong?

![Diagram showing a landscape of loss vs. a weight in W]
Setting the learning rate

Typical loss

The effects of step size (or “learning rate”)

(more on this later)

Slide from Karpathy 2016
Classification: Overview

Training Images

Test Image
Classification: Overview

Training Images

Gradient Descent

Test Image
Classification: Overview

Training Images

Training Labels

Gradient Descent

Test Image
Classification: Overview

Training Images → Training Labels → Gradient Descent → Trained Classifier

Test Image
Classification: Overview

- Training Images
- Training Labels
  - Gradient Descent
  - Trained Classifier
- Test Image
Classification: Overview

Training Images

Training Labels

Gradient Descent

Trained Classifier

Test Image

Prediction: “outdoor”
Neural Networks

(First we’ll cover Neural Nets, then build up to Convolutional Neural Nets)
Feature hierarchy with ConvNets
End-to-end models

[Zeiler and Fergus, “Visualizing and Understanding Convolutional Networks”, 2013]
Learning Feature Hierarchy

- Learn hierarchy
- All the way from pixels $\rightarrow$ classifier
- One layer extracts features from output of previous layer
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:

\[ f(x_i, W, b) = Wx_i + b \]
1 Layer Neural Net

Block Diagram:

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

(Input) \hspace{1cm} (class scores)
1 Layer Neural Net

Block Diagram:

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

(Input) \hspace{1cm} (class scores)

Expanded Block Diagram:

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

M classes

D features
1 example
1 Layer Neural Net

Block Diagram:

\[
\begin{align*}
x & \rightarrow Wx + b \\
& \rightarrow f
\end{align*}
\]

(Input) (class scores)

Expanded Block Diagram:

\[
\begin{align*}
W & \\
D & \\
1 &
\end{align*}
\]

\[
\begin{align*}
x & \\
D & \\
1 &
\end{align*}
\]

\[
\begin{align*}
b & \\
1 &
\end{align*}
\]

\[
\begin{align*}
M & = f \\
M &
\end{align*}
\]

NumPy:

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

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.

$$X^{(1)} \cdot W^{(1)} + b^{(1)} = F^{(1)}$$

- $M$ classes
- $D$ features
- $N$ examples
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.

\[
\begin{align*}
X & \quad W & + & \begin{pmatrix} b \\ \vdots \\ b \end{pmatrix} & = & F \\
N \times D & \quad M \times D & \quad 1 & \quad N \times 1 & \quad M & \quad N \\
M & \quad D & \quad M & \quad M & \quad M \\
\end{align*}
\]

**Note:** Often, if the weights are transposed, they are still called “W”.

\( M \) classes
\( D \) features
\( N \) examples
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

<table>
<thead>
<tr>
<th>X</th>
<th>D</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each row is one input example

<table>
<thead>
<tr>
<th>W</th>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

Each column is the weights for one class

<table>
<thead>
<tr>
<th>F</th>
<th>M</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

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:

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

Doesn’t work — why?
1 Layer Neural Net

Implementing this with NumPy:

First attempt — let’s try this:

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

N x D  M

D x M

Doesn’t work — why?
1 Layer Neural Net

Implementing this with NumPy:

First attempt — let’s try this:

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

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

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:

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

- 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?

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

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

Implementing this with NumPy:

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

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
Implementing this with NumPy:

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

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]])

In [5]: b[:, np.newaxis]
Out[5]: array([[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[\text{np.newaxis}, :] + \text{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?

\[
\begin{align*}
x & \rightarrow W^{(1)}x + b^{(1)} & \rightarrow h & \rightarrow W^{(2)}h + b^{(2)} & \rightarrow f
\end{align*}
\]
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:

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

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

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

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

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

What happens if we reach this part?

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**

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

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

```python
In [25]: sigmoid(np.array([-1, 2, 50]))
Out[25]: array([ 0.26894142, 0.88079708, 1.00000000])
```
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.        ])

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?

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

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

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

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?

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

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

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

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

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

Saturation: the gradient is zero!
Nonlinearities

[Krizhevsky 2012] (AlexNet)
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:

```python
h1relu = 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:

\[
\text{ReLU}(h_1) = \max(h_1, 0)
\]

(b) Make a boolean mask where negative values are True, and then set those entries in \( h_1 \) to 0:

\[
\text{ReLU}(h_1) = h_1 \cdot \mathbb{1}_{h_1 > 0}
\]

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

\[
\text{ReLU}(h_1) = h_1 \cdot (h_1 \geq 0)
\]
2 Layer Neural Net
2 Layer Neural Net

Layer 1:

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

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

(Layer 1) (Nonlinearity) ("hidden activations")
A 2 Layer Neural Net

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

(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
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

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

\textbf{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

2 layers

- 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

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

\[\begin{align*}
    \text{(Input)} \quad x & \rightarrow \text{Function} \rightarrow h^{(1)} \rightarrow \text{Function} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f
\end{align*}\]
Neural Networks,
More generally

\[ x \xrightarrow{\text{Function}} h^{(1)} \xrightarrow{\text{Function}} h^{(2)} \xrightarrow{\text{Hidden Activation}} \cdots \xrightarrow{\text{Scores}} f \]
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

\( h(1) \)

\( \theta^{(1)} \)

\( x \rightarrow \text{Function} \rightarrow h^{(1)} \)

\( \theta^{(2)} \)

\( h^{(2)} \rightarrow \cdots \rightarrow f \)

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

\[ h(1) \]
\[ h(2) \]
\[ \theta^{(1)} \]
\[ \theta^{(2)} \]
\[ f \]
\[ L \]

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.