Convolutional Neural Networks

CS 4670
Sean Bell

http://brownsharpie.courtneygibbons.org/?p=90
Review: Setup

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

\[ \downarrow \]

\[ y \rightarrow L \]
- **Goal**: Find a value for parameters \((\theta^{(1)}, \theta^{(2)}, \ldots)\), so that the loss \((L)\) is small
Review: Setup

Toy Example:
Review: Setup

$$x \rightarrow W^{(1)}x + b^{(1)} \rightarrow h^{(1)} \rightarrow \theta^{(2)}$$

Function

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

$$y \rightarrow y$$

Loss

Toy Example:

$$L$$

A weight somewhere in the network $W^{(1)}_{12}$
Review: Setup

\[ W^{(1)}, b^{(1)} \]

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

\[ y \rightarrow L \]

Toy Example:

Loss

A weight somewhere in the network

\[ W^{(1)}_{12} \]
Review: Setup

\[ W^{(1)}, b^{(1)} \]

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

\[ \text{Function} \rightarrow h^{(2)} \rightarrow \ldots \rightarrow f \]

\[ y \]

\[ \rightarrow \]

\[ L \]

Toy Example:

\[ L \]

A weight somewhere in the network
Review: Setup

\[ W^{(1)}, \, b^{(1)} \]

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

\[ y \rightarrow L \]

**Toy Example:**

Loss

A weight somewhere in the network

\[ L \rightarrow W^{(1)}_{12} \]
Review: Setup

A weight somewhere in the network

Toy Example:

Loss

$W^{(1)}, b^{(1)}$ → $W^{(1)}x + b^{(1)}$ → $h^{(1)}$ → Function → $h^{(2)}$ → ... → $f$ → $L$

Loss

$\frac{\partial L}{\partial W^{(1)}_{12}}$
Review: Setup

$W^{(1)}, b^{(1)}$ → $W^{(1)}x + b^{(1)}$ → $h^{(1)}$ → Function → $h^{(2)}$ → ... → $f$

$y$ → $L$

Toy Example:

Loss: $L$

A weight somewhere in the network

$$\frac{\partial L}{\partial W^{(1)}_{12}}$$ (Gradient)
Review: Setup

\[ W^{(1)}, b^{(1)} \]
\[ x \rightarrow W^{(1)}x + b^{(1)} \rightarrow h^{(1)} \rightarrow \text{Function} \rightarrow h^{(2)} \rightarrow \ldots \rightarrow f \]
\[ y \rightarrow L\]

**Toy Example:**

\[ \frac{\partial L}{\partial W^{(1)}_{12}} \] (Gradient)

Loss

A weight somewhere in the network

Take a step
Review: Setup

$W^{(1)}, b^{(1)}$ → $h^{(1)}$ → $h^{(2)}$ → … → $f$

$y$ → $L$

How do we get the gradient? **Backpropagation**

A weight somewhere in the network

Toy Example:

$L$ → $\frac{\partial L}{\partial W^{(1)}_{12}}$ (Gradient)
Backprop

It’s just the chain rule
\( \frac{\partial L}{\partial \theta^{(n)}} \)

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

Layer \( n \)

Layer \( n + 1 \)

...
This is what we want for each layer

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

Layer \( n \)
This is what we want for each layer: 

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

To compute it, we need to propagate this gradient:

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

\[ \frac{\partial L}{\partial h^{(n-1)}} \]
This is what we want for each layer.

To compute it, we need to propagate this gradient.

For each layer:
This is what we want for each layer. To compute it, we need to propagate this gradient.

For each layer:

\[
\frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial \theta^{(n)}}
\]

What we want
This is what we want for each layer.

To compute it, we need to propagate this gradient.

For each layer:

\[
\frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial \theta^{(n)}}
\]
This is what we want for each layer: 

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

To compute it, we need to propagate this gradient:

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

For each layer:

\[
\frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial \theta^{(n)}}
\]

What we want is just the local gradient of layer \( n \).
Backprop

This is what we want for each layer

\[ \frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial \theta^{(n)}} \]

To compute it, we need to propagate this gradient

For each layer:

\[ \frac{\partial L}{\partial h^{(n-1)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial h^{(n-1)}} \]

What we want

This is just the local gradient of layer \( n \)
This is what we want for each layer

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

To compute it, we need to propagate this gradient

For each layer:

\[ \frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial \theta^{(n)}} \]

\[ \frac{\partial L}{\partial h^{(n-1)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial h^{(n-1)}} \]

What we want

This is just the local gradient of layer \( n \)
This is what we want for each layer.

To compute it, we need to propagate this gradient.

For each layer:

\[
\frac{\partial L}{\partial \theta^{(n)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial \theta^{(n)}}
\]

\[
\frac{\partial L}{\partial h^{(n-1)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial h^{(n-1)}}
\]

What we want is just the local gradient of layer \( n \).
For each layer, we compute:

\[
[\text{Propagated gradient to the left}] = [\text{Propagated gradient from right}] \cdot [\text{Local gradient}]
\]
Backprop

For each layer, we compute:

\[ \text{[Propagated gradient to the left]} = \text{[Propagated gradient from right]} \cdot \text{[Local gradient]} \]

(Can compute immediately)
For each layer, we compute:

\[
\begin{align*}
\text{[Propagated gradient to the left]} &= \text{[Propagated gradient from right]} \cdot \text{[Local gradient]} \\
(\text{Received during backprop}) &\quad (\text{Can compute immediately})
\end{align*}
\]
Backprop

Forward Propagation:

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

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

Backward Propagation:
Backprop

Forward Propagation:

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

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

\[ \theta^{(n)} \]

Backward Propagation:
Backprop

Forward Propagation:

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

Backward Propagation:

\[ \frac{\partial L}{\partial f} \leftarrow L \]
Forward Propagation:

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

Backward Propagation:

\[ \frac{\partial L}{\partial \theta^{(n)}} \rightarrow \text{Function} \]

\[ \frac{\partial L}{\partial f} \rightarrow L \]
Backprop

Forward Propagation:

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

Backward Propagation:

\[ \frac{\partial L}{\partial h^{(1)}} \rightarrow \cdots \rightarrow \frac{\partial L}{\partial \theta^{(n)}} \rightarrow L \]
Backprop

Forward Propagation:

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

Backward Propagation:

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

\[ \frac{\partial L}{\partial \theta^{(n)}} \rightarrow \frac{\partial L}{\partial f} \]
Backprop

It’s easy to write down the chain rule for higher dimensions — just add more subscripts and more summations
Backprop

It’s easy to write down the chain rule for higher dimensions — just add more subscripts and more summations

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

$x, h$ scalars

$(L$ is always scalar$)$
Backprop

It’s easy to write down the chain rule for higher dimensions — just add more subscripts and more summations.

\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial x} \quad x,h \text{ scalars}
\]

\[
\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j} \quad x,h \text{ 1D arrays (vectors)}
\]
Backprop

It’s easy to write down the chain rule for higher dimensions — just add more subscripts and more summations

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

\[ \frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j} \]

\[ \frac{\partial L}{\partial x_{ab}} = \sum_i \sum_j \frac{\partial L}{\partial h_{ij}} \frac{\partial h_{ij}}{\partial x_{ab}} \]

\(x, h\) scalars

\(L\) is always scalar

\(x, h\) 1D arrays (vectors)

\(x, h\) 2D arrays
Backprop

It’s easy to write down the chain rule for higher dimensions — just add more subscripts and more summations

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

\[
\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j}
\]

\[
\frac{\partial L}{\partial x_{ab}} = \sum_i \sum_j \frac{\partial L}{\partial h_{ij}} \frac{\partial h_{ij}}{\partial x_{ab}}
\]

\[
\frac{\partial L}{\partial x_{abc}} = \sum_i \sum_j \sum_k \frac{\partial L}{\partial h_{ijk}} \frac{\partial h_{ijk}}{\partial x_{abc}}
\]

\(x,h\) scalars

\(L\) is always scalar

\(x,h\) 1D arrays (vectors)

\(x,h\) 2D arrays

\(x,h\) 3D arrays
Example: Mean Subtraction
(for a single input)
Example: Mean Subtraction
(for a single input)

• Ok, so how do we actually derive the backwards pass? Let’s walk through an example together.
Example: Mean Subtraction
(for a single input)

- Ok, so how do we actually derive the backwards pass? Let’s walk through an example together.

- Example layer: mean subtraction:

\[ h_i = x_i - \frac{1}{D} \sum_{k} x_k \]
Example: Mean Subtraction
(for a single input)

• Ok, so how do we actually derive the backwards pass? Let’s walk through an example together.

• Example layer: mean subtraction:

\[ h_i = x_i - \frac{1}{D} \sum_k x_k \]  (here, “i” and “k” are channels)
Ok, so how do we actually derive the backwards pass? Let’s walk through an example together.

Example layer: mean subtraction:

$$h_i = x_i - \frac{1}{D} \sum_k x_k$$  

(here, “i” and “k” are channels)

For backprop, we just need the local derivative.
Example: Mean Subtraction
(for a single input)
Example: Mean Subtraction
(for a single input)

- Forward: \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]
Example: Mean Subtraction
(for a single input)

• Forward:  \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

• Taking the derivative of the layer:
Example: Mean Subtraction
(for a single input)

- Forward: \( h_i = x_i - \frac{1}{D} \sum_k x_k \)

- Taking the derivative of the layer: \( \frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D} \)
Example: Mean Subtraction
(for a single input)

• Forward: \( h_i = x_i - \frac{1}{D} \sum_k x_k \)

• Taking the derivative of the layer:
  \[
  \frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}
  \]

\[
\delta_{ij} = \begin{cases} 
  1 & i = j \\
  0 & \text{else}
\end{cases}
\]
Example: Mean Subtraction
(for a single input)

- **Forward:** \( h_i = x_i - \frac{1}{D} \sum_k x_k \)

- **Taking the derivative of the layer:**
  \[
  \frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}
  \]

\[
\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j} \quad \text{(backprop aka chain rule)}
\]

\[
\delta_{ij} = \begin{cases} 
1 & i = j \\
0 & \text{else}
\end{cases}
\]
Example: Mean Subtraction
(for a single input)

- Forward: \( h_i = x_i - \frac{1}{D} \sum_k x_k \)
- Taking the derivative of the layer:
  \[
  \frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}
  \]

\[
\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j} \quad \text{(backprop aka chain rule)}
\]

\[
= \sum_i \frac{\partial L}{\partial h_i} \left( \delta_{ij} - \frac{1}{D} \right)
\]

\[
\delta_{ij} = \begin{cases} 
1 & \text{if } i = j \\
0 & \text{else}
\end{cases}
\]
Example: Mean Subtraction
(for a single input)

- **Forward:** \( h_i = x_i - \frac{1}{D} \sum_k x_k \)

- **Taking the derivative of the layer:**
  \[
  \frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}
  \]
  (backprop aka chain rule)

\[
\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j}
\]

\[
= \sum_i \frac{\partial L}{\partial h_i} \left( \delta_{ij} - \frac{1}{D} \right)
\]

\[
= \sum_i \frac{\partial L}{\partial h_i} \delta_{ij} - \frac{1}{D} \sum_i \frac{\partial L}{\partial h_i}
\]

\[
\delta_{ij} = \begin{cases} 
  1 & i = j \\
  0 & \text{else}
\end{cases}
\]
Example: Mean Subtraction
(for a single input)

- Forward: \( h_i = x_i - \frac{1}{D} \sum_k x_k \)
- Taking the derivative of the layer: \( \frac{\partial h}{\partial x_j} = \delta_{ij} - \frac{1}{D} \)

\[
\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j} \quad \text{(backprop aka chain rule)}
\]

\[
= \sum_i \frac{\partial L}{\partial h_i} \left( \delta_{ij} - \frac{1}{D} \right)
\]

\[
= \sum_i \frac{\partial L}{\partial h_i} \delta_{ij} - \frac{1}{D} \sum_i \frac{\partial L}{\partial h_i}
\]

\[
= \frac{\partial L}{\partial h_j} - \frac{1}{D} \sum_i \frac{\partial L}{\partial h_i}
\]

\[
\delta_{ij} = \begin{cases} 
1 & i = j \\
0 & \text{else}
\end{cases}
\]
Example: Mean Subtraction  
(for a single input)

• Forward: \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

• Taking the derivative of the layer: \[ \frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D} \]

\[
\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j} \quad \text{(backprop aka chain rule)}
\]

\[
= \sum_i \frac{\partial L}{\partial h_i} \left( \delta_{ij} - \frac{1}{D} \right)
\]

\[
= \sum_i \frac{\partial L}{\partial h_i} \delta_{ij} - \frac{1}{D} \sum_i \frac{\partial L}{\partial h_i}
\]

\[
= \frac{\partial L}{\partial h_j} - \frac{1}{D} \sum_i \frac{\partial L}{\partial h_i}
\]

\[
\delta_{ij} = \begin{cases} 
1 & i = j \\
0 & \text{else}
\end{cases}
\]

Done!
Example: Mean Subtraction
(for a single input)

\[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

\[ \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D} \sum_k \frac{\partial L}{\partial h_k} \]
Example: Mean Subtraction
(for a single input)

- **Forward:** \( h_i = x_i - \frac{1}{D} \sum_k x_k \)
- **Backward:** \( \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D} \sum_k \frac{\partial L}{\partial h_k} \)
Example: Mean Subtraction
(for a single input)

- Forward: \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

- Backward: \[ \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D} \sum_k \frac{\partial L}{\partial h_k} \]

- In this case, they’re identical operations!
Example: Mean Subtraction
(for a single input)

- Forward: \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]
- Backward: \[ \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D} \sum_k \frac{\partial L}{\partial h_k} \]

- In this case, they’re identical operations!
- Usually the forwards pass and backwards pass are similar but not the same.
Example: Mean Subtraction
(for a single input)

- Forward: \( h_i = x_i - \frac{1}{D} \sum_k x_k \)
- Backward: \( \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D} \sum_k \frac{\partial L}{\partial h_k} \)

- In this case, they’re identical operations!
- Usually the forwards pass and backwards pass are similar but not the same.
- Derive it by hand, and check it numerically
Example: Mean Subtraction  
(for a single input)

- Forward:  \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

Let’s code this up in NumPy:
Example: Mean Subtraction
(for a single input)

- Forward: \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

Let’s code this up in NumPy:

```python
def forward(X):
    return X - np.mean(X, axis=1)
```
Example: Mean Subtraction  
(for a single input)

- Forward: \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

Let's code this up in NumPy:

```python
def forward(X):
    return X - np.mean(X, axis=1)
```
Example: Mean Subtraction
(for a single input)

- Forward: \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

Let's code this up in NumPy:

```python
def forward(X):
    return X - np.mean(X, axis=1)
```

You need to broadcast properly:

```python
def forward(X):
    return X - np.mean(X, axis=1)[:, np.newaxis]
```
Example: Mean Subtraction
(for a single input)

• Forward: \[ h_i = x_i - \frac{1}{D} \sum_k x_k \]

Let's code this up in NumPy:

```python
def forward(X):
    return X - np.mean(X, axis=1)
```

You need to broadcast properly:

```python
def forward(X):
    return X - np.mean(X, axis=1)[:, np.newaxis]
```

This also works:

```python
def forward(X):
    return X - np.mean(X, axis=1, keepdims=True)
```
Example: Mean Subtraction
(for a single input)

The backward pass is easy:

```python
def backward(dh):
    return forward(dh)
```

(Remember they’re usually not the same)
Example: Softmax (for N inputs)

Let’s assume we are using Softmax and Cross-entropy loss (together this is often called “Softmax loss”)
Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss (together this is often called "Softmax loss")

\[ y_i \rightarrow \cdots \rightarrow f_i \rightarrow \text{Softmax} \rightarrow p_i \rightarrow \text{Cross-Entropy} \rightarrow L_i \]
Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss (together this is often called “Softmax loss”)

(ground truth labels)

\( y_i \)

\( x_i \rightarrow \cdots \rightarrow f_i \rightarrow \text{Softmax} \rightarrow p_i \rightarrow \text{Cross-Entropy} \rightarrow L_i \)

(input) (scores) (probabilities) (loss)
Example: Softmax (for N inputs)

Let’s assume we are using Softmax and Cross-entropy loss (together this is often called “Softmax loss”)

(y_i) (ground truth labels)

x_i → ... → f_i → Softmax → p_i → Cross-Entropy → L_i

(x_i) (input) (f_i) (scores) (p_i) (probabilities) (L_i) (loss)
Example: Softmax (for N inputs)

Let’s assume we are using Softmax and Cross-entropy loss (together this is often called “Softmax loss”)

\[
p_{i,j} = \frac{e^{f_{i,j}}}{\sum_{k} e^{f_{i,k}}} \]  

Softmax

(ground truth labels)  

\(y_i\)  

(input)  

\(x_i \rightarrow \cdots \rightarrow f_i \rightarrow \text{Softmax} \rightarrow p_i \rightarrow \text{Cross-Entropy} \rightarrow L_i\)  

(scores)  

(proBABILITIES)  

(loss)  

(here, “i” are different examples)
Example: Softmax (for N inputs)

Let’s assume we are using Softmax and Cross-entropy loss (together this is often called “Softmax loss”)

$y_i$ (ground truth labels)

$x_i \rightarrow \ldots \rightarrow f_i \rightarrow \text{Softmax} \rightarrow p_i \rightarrow \text{Cross-Entropy} \rightarrow L_i$

(input) (scores) (probabilities) (loss)

$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$

(Softmax)

$L_i = -\log p_{i,y_i}$

(Cross-entropy)
Example: Softmax (for N inputs)

Let’s assume we are using Softmax and Cross-entropy loss (together this is often called “Softmax loss”)

**Ground truth labels**

\[ y_i \]

**Input**

\[ x_i \rightarrow \ldots \rightarrow f_i \]

**Scores**

\[ \text{Softmax} \rightarrow p_i \]

**Probabilities**

\[ \text{Cross-Entropy} \rightarrow L_i \]

**Loss**

\[ L_i = - \log p_{i,y_i} \]

\[ L = \frac{1}{N} \sum_i L_i \]

**Softmax**

\[ p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}} \]

**Cross-entropy loss**

\[ L_i = - \log p_{i,y_i} \]

**Average over examples**
Example: Softmax (for N inputs)

\[ x_i \rightarrow \cdots \rightarrow f_i \rightarrow \text{Softmax} \rightarrow p_i \rightarrow \text{Cross-Entropy} \rightarrow L_i \]
Example: Softmax (for N inputs)

Derivative:

\[
\frac{\partial L}{\partial f_{i,j}} = \frac{p_{i,j} - t_{i,j}}{N}
\]
Example: Softmax (for N inputs)

\[ \frac{\partial L}{\partial f_{i,j}} = \frac{p_{i,j} - t_{i,j}}{N} \]

where \( t_i = [0 \ldots 1 \ldots 0] \) (Entry \( y_i \) set to 1)

Derivative:

\[ y_i \]

\[ x_i \rightarrow \cdots \rightarrow f_i \rightarrow \text{Softmax} \rightarrow p_i \rightarrow \text{Cross-Entropy} \rightarrow L_i \]
Example: Softmax (for N inputs)

\[
\frac{\partial L}{\partial f_{i,j}} = \frac{p_{i,j} - t_{i,j}}{N}
\]

where \( t_i = [0 \ldots 1 \ldots 0] \) (Entry \( y_i \) set to 1)
Example: Softmax (for N inputs)

\[
\frac{\partial L}{\partial f_{i,j}} = \frac{p_{i,j} - t_{i,j}}{N}
\]

where \( t_i = [0 \ldots 1 \ldots 0] \) (Entry \( y_i \) set to 1)

(Try deriving this — it’s tricky but not too hard)
Example: Softmax (for N inputs)

\[ y_i \]

\[ x_i \rightarrow \cdots \rightarrow f_i \rightarrow \text{Softmax} \rightarrow p_i \rightarrow \text{Cross-Entropy} \rightarrow L_i \]

Derivative:

\[
\frac{\partial L}{\partial f_{i,j}} = \frac{p_{i,j} - t_{i,j}}{N}
\]

where \( t_i = [0 \ldots 1 \ldots 0] \)

(Entry \( y_i \) set to 1)

(Try deriving this — it’s tricky but not too hard)

Now we can continue backpropagating to the layer before “f”
Example: Softmax (for N inputs)

Let's code this up in NumPy:

```python
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$
Example: Softmax (for N inputs)

Let’s code this up in NumPy:

```python
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Doesn’t work — what’s the problem this time?
Example: Softmax (for N inputs)

Let’s code this up in NumPy:

```python
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Doesn’t work — what’s the problem this time?
- What if there is the value 1000 appears in “f”? 
Example: Softmax (for N inputs)

Let’s code this up in NumPy:

```python
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

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

- What if there is the value 1000 appears in “f”?

  Overflow —> we get inf/inf = NaN
Example: Softmax (for N inputs)

Let’s code this up in NumPy:

```
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

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

  *Overflow* —> *we get inf/inf = NaN*

- What if the largest value is -1000?
Example: Softmax (for N inputs)

Let’s code this up in NumPy:

```python
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

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

- What if there is the value 1000 appears in “f”?
  
  *Overflow* —> we get \( \text{inf/inf} = \text{NaN} \)

- What if the largest value is -1000?
  
  *Underflow* —> we get 0/0 = NaN
Example: Softmax (for N inputs)

Let's code this up in NumPy:

```python
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

**Doesn't work** — what's the problem this time?
- What if there is the value 1000 appears in "f"?
  
  *Overflow* —> *we get inf/inf = NaN*

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

This expression is numerically unstable
Example: Softmax (for N inputs)
Example: Softmax (for N inputs)

Observation: subtracting a constant does not change “p”: 
Example: Softmax (for N inputs)

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

\[ p_{i,j} = \frac{e^{f_{i,j} - C}}{\sum_k e^{f_{j,k} - C}} = \frac{e^{-C} e^{f_{i,j}}}{\sum_k e^{-C} e^{f_{i,k}}} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}} \]
Example: Softmax (for N inputs)

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

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

If we choose “C” to be the max, then it works:
Example: Softmax (for N inputs)

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

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

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

- If a large value appears in “f”, then that value will become 1 and all others will be 0 (avoiding overflow)
Example: Softmax (for N inputs)

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

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

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

- If a large value appears in “f”, then that value will become 1 and all others will be 0 (avoiding overflow)
- If all values in “f” are large negative, then they will be shifted up towards 0 (avoiding underflow)
Observation: subtracting a constant does not change “p”:

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

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

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

```python
def softmax(f):
    exp_f = np.exp(f - np.max(f, axis=1, keepdims=True))
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```
What about the weights?

To get the derivative of the weights, use the chain rule again!
What about the weights?

To get the derivative of the weights, use the chain rule again!

Example: 2D weights, 1D bias, 1D hidden activations:
What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:

\[
x \rightarrow \text{Layer} \rightarrow h \quad h = h(x; W)
\]
What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:

\[
\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}}
\]
What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:

\[
\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}} \quad \frac{\partial L}{\partial b_i} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial b_i}
\]

\[
x \rightarrow \text{Layer} \rightarrow h \quad h = h(x; W)
\]
What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:

\[
\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}} \quad \frac{\partial L}{\partial b_i} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial b_i}
\]

(the number of subscripts and summations changes depending on your layer and parameter sizes)
What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:

\[
W, b \\
x \rightarrow \text{Layer} \rightarrow h \quad h = h(x; W)
\]

\[
\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}} \quad \frac{\partial L}{\partial b_i} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial b_i}
\]

(the number of subscripts and summations changes depending on your layer and parameter sizes)

**HW2:** you will derive this for various layers.
Recap

Forward Propagation:

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

Forward Propagation:

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

Backward Propagation:
Recap

Forward Propagation:

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

Backward Propagation:
Recap

Forward Propagation:

\[
\begin{align*}
\theta^{(1)} & \quad \text{Function} \quad h^{(1)} \quad \cdots \quad \text{Function} \\
x & \rightarrow \quad h^{(1)} & \rightarrow \quad \cdots & \rightarrow \quad f & \rightarrow \quad L
\end{align*}
\]

Backward Propagation:

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

Forward Propagation:

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

Backward Propagation:

\[ \frac{\partial L}{\partial \theta^{(n)}} \rightarrow \frac{\partial L}{\partial f} \rightarrow L \]
Recap

Forward Propagation:

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

Backward Propagation:

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

Forward Propagation:

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

Backward Propagation:

\[ \frac{\partial L}{\partial x} \leftarrow \frac{\partial L}{\partial \theta^{(1)}} \leftarrow \frac{\partial L}{\partial h^{(1)}} \leftarrow \ldots \leftarrow \frac{\partial L}{\partial \theta^{(n)}} \leftarrow \frac{\partial L}{\partial f} \leftarrow L \]
Questions?
30s cat picture break

http://stylonica.com/cat-pictures/
CNNs

It’s just neural networks with 3D activations
What shape should the activations have?

- The input is an image, which is 3D (RGB channel, height, width)
What shape should the activations have?

- The input is an image, which is 3D (RGB channel, height, width)
- We could flatten it to a 1D vector, but then we lose structure
What shape should the activations have?

- The input is an image, which is 3D (RGB channel, height, width)
- We could flatten it to a 1D vector, but then we lose structure
- What about keeping everything in 3D?
3D Activations

before:

Figure: Andrej Karpathy
3D Activations

before:

now:

Figure: Andrej Karpathy
3D Activations

All Neural Net activations arranged in 3 dimensions:

Figure: Andrej Karpathy
3D Activations

All Neural Net activations arranged in 3 dimensions:

For example, a CIFAR-10 image is a 3x32x32 volume (3 depth — RGB channels, 32 height, 32 width)

Figure: Andrej Karpathy
3D Activations

1D Activations:

Figure: Andrej Karpathy
3D Activations

**1D Activations:**

Figure: Andrej Karpathy
3D Activations

- The input is 3x32x32
- This neuron depends on a 3x5x5 chunk of the input
- The neuron also has a 3x5x5 set of weights and a bias (scalar)

Figure: Andrej Karpathy
3D Activations

Example: consider the region of the input \( x^r \)

With output neuron \( h^r \)

Figure: Andrej Karpathy
3D Activations

Example: consider the region of the input \( x^r \)

With output neuron \( h^r \)

Then the output is:

\[
h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b
\]
Example: consider the region of the input $x^r$.

With output neuron $h^r$.

Then the output is:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

Sum over 3 axes.

Figure: Andrej Karpathy
3D Activations

Figure: Andrej Karpathy
3D Activations

Figure: Andrej Karpathy
3D Activations

With 2 output neurons

\[
\begin{align*}
    h_1^r &= \sum_{ijk} x_{ijk} W_{1ijk} + b_1 \\
    h_2^r &= \sum_{ijk} x_{ijk} W_{2ijk} + b_2
\end{align*}
\]

Figure: Andrej Karpathy
3D Activations

With 2 output neurons

\[ h^r_1 = \sum_{ijk} x^r_{ijk} W_{1ijk} + b_1 \]

\[ h^r_2 = \sum_{ijk} x^r_{ijk} W_{2ijk} + b_2 \]
3D Activations

Figure: Andrej Karpathy
3D Activations

We can keep adding more outputs

These form a column in the output volume: $[\text{depth} \times 1 \times 1]$

Figure: Andrej Karpathy
3D Activations

Each neuron has its own 3D filter and own (scalar) bias.

We can keep adding more outputs.

These form a column in the output volume: $[\text{depth} \times 1 \times 1]$

Figure: Andrej Karpathy
3D Activations

Now repeat this across the input

$D$ sets of weights (also called filters)

Figure: Andrej Karpathy
3D Activations

Now repeat this across the input

Weight sharing:
Each filter shares the same weights (but each depth index has its own set of weights)

Figure: Andrej Karpathy
3D Activations

$D$ sets of weights
(also called filters)

Figure: Andrej Karpathy
3D Activations

With weight sharing, this is called **convolution**

$D$ sets of weights (also called filters)

*Figure: Andrej Karpathy*
3D Activations

With weight sharing, this is called convolution

Without weight sharing, this is called a locally connected layer

$D$ sets of weights (also called filters)

Figure: Andrej Karpathy
3D Activations

One set of weights gives one slice in the output.

To get a 3D output of depth $D$, use $D$ different filters.

In practice, CNNs use many filters (~64 to 1024)
3D Activations

One set of weights gives one slice in the output.

To get a 3D output of depth $D$, use $D$ different filters.

In practice, CNNs use many filters (~64 to 1024).

All together, the weights are 4-dimensional:
(output depth, input depth, kernel height, kernel width)
3D Activations

Let's code this up in NumPy

```python
out[n, 0, r, c] =
```
3D Activations

Let's code this up in NumPy

```
out[n, 0, r, c] =
```
3D Activations

Let’s code this up in NumPy

\[
\text{out}[n, 0, r, c] = \text{first filter} \]

\text{an n}^{\text{th}} \text{ example}
3D Activations

Let's code this up in NumPy

\[
\text{out}[n, 0, r, c] =
\]

- \(n^{th}\) example
- first filter
- output position

a hidden neuron in next layer
3D Activations

Let's code this up in NumPy

```python
out[n, 0, r, c] = np.sum(out[n, 0, r, c])
```
Let’s code this up in NumPy

\[
\text{out}[n, 0, r, c] = \text{np.sum}(X[n, :, r0:r1, c0:c1])
\]
3D Activations

Let’s code this up in NumPy

\[ \text{out}[n, 0, r, c] = \text{np.sum}(X[n, :, r0:r1, c0:c1]) \]
3D Activations

Let's code this up in NumPy

```python
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1]
```

- **n<sup>th</sup> example**
- **output position**
- **first filter**
- **all input channels**
3D Activations

Let's code this up in NumPy

\[
\text{out}[n, 0, r, c] = \text{np.sum}(X[n, :, r_0:r_1, c_0:c_1])
\]

- \(n^\text{th}\) example
- First filter
- Output position
- Input region
- All input channels
3D Activations

Let's code this up in NumPy

\[
\text{out}[n, 0, r, c] = \text{np.sum}(X[n, :, r0:r1, c0:c1] \times W[0, :, :, :]) + b[0]
\]

-的第一步filter
- 输出位置
- 第n个例子
- 输入区域
- 所有输入通道
- 第n个例子
3D Activations

Let's code this up in NumPy

\[
\text{out}[n, 0, r, c] = \text{np.sum}(X[n, :, r0:r1, c0:c1] \ast W[0, :, :, :]) + b[0]
\]

first filter

output position

n^{th} example

input region

all input channels

n^{th} example

first filter
3D Activations

Let's code this up in NumPy

```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1] * W[0, :, :, :]) + b[0]
```
3D Activations

Let's code this up in NumPy

\[
\text{out}[n, 0, r, c] = \text{np.sum}(X[n, :, r0:r1, c0:c1] * W[0, :, :, :]) + b[0]
\]
3D Activations

Let's code this up in NumPy

\[
\text{out}[n, 0, r, c] = \text{np.sum}(X[n, :, r0:r1, c0:c1] \times W[0, :, :, :]) + b[0]
\]
3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

(32 filters, each 3x5x5)

Figure: Andrej Karpathy
3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

(32 filters, each 3x5x5)

one filter = one depth slice (or activation map)

Figure: Andrej Karpathy
3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

(32 filters, each 3x5x5)

Figure: Andrej Karpathy
3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

(32 filters, each 3x5x5)

Figure: Andrej Karpathy
Questions?
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output.
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output.
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output.
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output.
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output.
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output.
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output.

Recall that at each position, we are doing a 3D sum:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

(channel, row, column)
Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2
Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2
Convolution: Stride

But we can also convolve with a **stride**, e.g. \( \text{stride} = 2 \)
Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2

- Notice that with certain strides, we may not be able to cover all of the input
Convolution: Stride

But we can also convolve with a **stride**, e.g. **stride = 2**

- Notice that with certain strides, we may not be able to cover all of the input
- The output is also half the size of the input
Convolution: Padding

We can also pad the input with zeros. Here, **pad = 1, stride = 2**

![Input and Output Diagram]
We can also pad the input with zeros. Here, \( \text{pad} = 1, \text{stride} = 2 \)
Convolution: Padding

We can also pad the input with zeros. Here, **pad = 1, stride = 2**
Convolution: Padding

We can also pad the input with zeros. Here, $\text{pad} = 1$, $\text{stride} = 2$
Convolution:
How big is the output?
Convolution: How big is the output?
Convolution:
How big is the output?

stride $s$

kernel $k$
Convolution: How big is the output?

stride $s$

width $w_{\text{in}}$

kernel $k$
Convolution: How big is the output?

stride $s$

kernel $k$

width $w_{in}$
Convolution:
How big is the output?

In general, the output has size:

$$w_{out} = \left\lfloor \frac{w_{in} + 2p - k}{s} \right\rfloor + 1$$
Convolution:
How big is the output?

Example: $k=3$, $s=1$, $p=1$
Convolution: How big is the output?

Example: $k=3$, $s=1$, $p=1$

$$w_{\text{out}} = \left\lfloor \frac{w_{\text{in}} + 2p - k}{s} \right\rfloor + 1$$

$$= \left\lfloor \frac{w_{\text{in}} + 2 - 3}{1} \right\rfloor + 1$$

$$= w_{\text{in}}$$
Convolution:
How big is the output?

Example: \( k=3, s=1, p=1 \)

\[
w_{\text{out}} = \left\lfloor \frac{\text{\textit{w}_{\text{in}}} + 2p - k}{s} \right\rfloor + 1
\]

\[
= \left\lfloor \frac{\text{\textit{w}_{\text{in}}} + 2 - 3}{1} \right\rfloor + 1
\]

\[
= \text{\textit{w}_{\text{in}}}
\]

VGGNet [Simonyan 2014] uses filters of this shape.
Max Pooling

For most CNNs, **convolution** is often followed by **pooling**:
Max Pooling

For most CNNs, **convolution** is often followed by **pooling**:  
- Creates a smaller representation while retaining the most important information

*Figure: Andrej Karpathy*
Max Pooling

For most CNNs, **convolution** is often followed by **pooling**:
- Creates a smaller representation while retaining the most important information
- The “max” operation is the most common

*Figure: Andrej Karpathy*
Max Pooling

For most CNNs, **convolution** is often followed by **pooling**:  
- Creates a smaller representation while retaining the most important information  
- The “max” operation is the most common  
- Why might “avg” be a poor choice?

*Figure: Andrej Karpathy*
Max Pooling

Single depth slice

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

max pool with 2x2 filters and stride 2

<table>
<thead>
<tr>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure: Andrej Karpathy
Max Pooling

What’s the backprop rule for max pooling?

Figure: Andrej Karpathy
Max Pooling

What’s the backprop rule for max pooling?

- In the forward pass, store the index that took the max
What’s the backprop rule for max pooling?

- In the forward pass, store the index that took the max
- The backprop gradient is the input gradient at that index
Example CNN

CONV ReLU CONV ReLU POOL

Figure: Andrej Karpathy
Example CNN

Figure: Andrej Karpathy
Example CNN
Example CNN

10x3x3 conv filters, stride 1, pad 1
2x2 pool filters, stride 2

Figure: Andrej Karpathy
Questions?