Topics:
• Convolutional Neural Networks
The connectivity in linear layers doesn’t always make sense

How many parameters?
- $M \times N$ (weights) + $N$ (bias)

Hundreds of millions of parameters for just one layer

More parameters => More data needed

Is this necessary?
Image features are spatially localized!

- Smaller features repeated across the image
  - Edges
  - Color
  - Motifs (corners, etc.)

- No reason to believe one feature tends to appear in one location vs. another (stationarity)

Can we induce a bias in the design of a neural network layer to reflect this?
Each node only receives input from $K_1 \times K_2$ window (image patch)

- Region from which a node receives input from is called its **receptive field**

**Advantages:**

- Reduce parameters to $(K_1 \times K_2 + 1) \times N$ where $N$ is number of output nodes
- Explicitly maintain spatial information

Do we need to learn location-specific features?

**Idea 1: Receptive Fields**
Nodes in different locations can share features

- No reason to think same feature (e.g. edge pattern) can’t appear elsewhere
- Use same weights/parameters in computation graph (shared weights)

Advantages:

- Reduce parameters to \((K_1 \times K_2 + 1)\)
- Explicitly maintain spatial information

Idea 2: Shared Weights
We can learn many such features for this one layer

- Weights are not shared across different feature extractors

- **Parameters:** $(K_1 \times K_2 + 1) \times M$ where $M$ is number of features we want to learn

**Idea 3: Learn Many Features**
This operation is **extremely common** in electrical/computer engineering!

Convolution

From https://en.wikipedia.org/wiki/Convolution
This operation is extremely common in electrical/computer engineering!

Convolution

From https://en.wikipedia.org/wiki/Convolution
This operation is **extremely common** in electrical/computer engineering!

In mathematics and, in particular, functional analysis, **convolution** is a mathematical operation on two functions $f$ and $g$ producing a third function that is typically viewed as a modified version of one of the original functions, giving the area overlap between the two functions as a function of the amount that one of the original functions is translated.

Convolution is similar to **cross-correlation**.

It has **applications** that include probability, statistics, computer vision, image and signal processing, electrical engineering, and differential equations.

*From https://en.wikipedia.org/wiki/Convolution*
Notation: \( F \otimes (G \otimes I) = (F \otimes G) \otimes I \)

1D Convolution

\[ y_k = \sum_{n=0}^{N-1} h_n \cdot x_{k-n} \]

\[
\begin{align*}
y_0 &= h_0 \cdot x_0 \\
y_1 &= h_1 \cdot x_0 + h_0 \cdot x_1 \\
y_2 &= h_2 \cdot x_0 + h_1 \cdot x_1 + h_0 \cdot x_2 \\
y_3 &= h_3 \cdot x_0 + h_2 \cdot x_1 + h_1 \cdot x_2 + h_0 \cdot x_3 \\
&\vdots
\end{align*}
\]

\[
K = \begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1 \\
\end{bmatrix}
\]

2D Convolution

2D Discrete Convolution
2D Discrete Convolution

\[ K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \]

Image

Kernel (or filter)

Output / filter / feature map
We will make this convolution operation a **layer** in the neural network

- Initialize kernel values randomly and optimize them!
- These are our parameters (plus a bias term per filter)
1. Flip kernel (rotate 180 degrees)

2. Stride along image
Mathematics of Discrete 2D Convolution

\[
y(r, c) = (x \ast k)(r, c) = \sum_{a=-\frac{H-1}{2}}^{\frac{H-1}{2}} \sum_{b=-\frac{W-1}{2}}^{\frac{W-1}{2}} x(a, b) k(r - a, c - b)
\]

\[
\begin{pmatrix}
-\frac{H-1}{2} & -\frac{W-1}{2}
\end{pmatrix}
\]

\[
H = 5
\]

\[
W = 5
\]

\[
(0,0)
\]

\[
k_1 = 3
\]

\[
(0,0)
\]

\[
k_2 = 3
\]

\[
(k_1 - 1, k_2 - 1)
\]

\[
y(0, 0) = x(-2, -2)k(2, 2) + x(-2, -1)k(2, 1) + x(-2, 0)k(2, 0) + x(-2, 1)k(2, -1) + x(-2, 2)k(2, -2) + ...
\]
\[ y(r, c) = (x \ast k)(r, c) = \sum_{a=-\frac{k_1-1}{2}}^{\frac{k_1-1}{2}} \sum_{b=-\frac{k_2-1}{2}}^{\frac{k_2-1}{2}} x(r-a, c-b) k(a, b) \]

\[ k_1 = 3 \]

\[ k_2 = 3 \]

Centering Around the Kernel

\[ W = 5 \]

\[ (H - 1, W - 1) \]
As we have seen:

- **Convolution**: Start at end of kernel and move back
- **Cross-correlation**: Start in the beginning of kernel and move forward (same as for image)

An **intuitive interpretation** of the relationship:

- Take the kernel, and rotate 180 degrees along center (sometimes referred to as “flip”)
- Perform cross-correlation
- (Just dot-product filter with image!)

\[
K = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]

\[
K' = \begin{bmatrix}
9 & 8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1 \\
\end{bmatrix}
\]
\[ y(r, c) = (x * k)(r, c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r+a, c+b) k(a,b) \]

Since we will be learning these kernels, this change does not matter!
Cross-Correlation

\[
X(0:2,0:2) = \begin{bmatrix}
200 & 150 & 150 \\
100 & 50 & 100 \\
25 & 25 & 10
\end{bmatrix}
\quad K' = \begin{bmatrix}
1 & 0 & -1 \\
2 & 0 & -2 \\
1 & 0 & -1
\end{bmatrix}
\]

\[
X(0:2,0:2) \cdot K' = 65 + \text{bias}
\]

Dot product (element-wise multiply and sum)
Convolution and Cross-Correlation
Convolution and Cross-Correlation
Convolution and Cross-Correlation
Convolution and Cross-Correlation
Why Bother with Convolutions?

Convolutions are just **simple linear operations**

**Why bother** with this and not just say it’s a linear layer with small receptive field?

- There is a **duality** between them during backpropagation
- Convolutions have **various mathematical properties** people care about
- This is **historically** how it was inspired
Input & Output Sizes
Convolution operations have several hyper-parameters

- `in_channels (int)` – Number of channels in the input image
- `out_channels (int)` – Number of channels produced by the convolution
- `kernel_size (int or tuple)` – Size of the convolving kernel
- `stride (int or tuple, optional)` – Stride of the convolution. Default: 1
- `padding (int or tuple, optional)` – Zero-padding added to both sides of the input. Default: 0
- `padding_mode (string, optional)` – 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Output size of vanilla convolution operation is \((H - k_1 + 1) \times (W - k_2 + 1)\)

- This is called a “valid” convolution and only applies kernel within image

\[H = 5\]
\[W = 5\]
\[(0, 0)\]
\[(0, 0)\]
\[k_1 = 3\]
\[k_2 = 3\]
\[k_1 - 1, k_2 - 1\]

\((H - 1, W - 1)\)
We can **pad the images** to make the output the same size:

- Zeros, mirrored image, etc.
- Note padding often refers to pixels added to one size \((P = 1\) here)
We can move the filter along the image using larger steps (**stride**)

- This can potentially result in **loss of information**
- Can be used for **dimensionality reduction** (not recommended)

\[
\frac{H - k_1}{2} + 1 \quad \text{Stride} = 2 \text{ (every other pixel)}
\]

\[
\frac{W - k_2}{2} + 1
\]
Stride can result in **skipped pixels**, e.g. stride of 3 for 5x5 input.
We have shown inputs as a one-channel image but in reality they have three channels (red, green, blue).

- In such cases, we have 3-channel kernels!
We have shown inputs as a one-channel image but in reality they have three channels (red, green, blue).

- In such cases, we have 3-channel kernels!

Similar to before, we perform element-wise multiplication between kernel and image patch, summing them up (dot product).

- Except with $k_1 \times k_2 \times 3$ values
We can have **multiple kernels per layer**

- We stack the feature maps together at the output

Number of channels in output is equal to *number of kernels*.

- Image: $H \times W$
- Kernels: $k_1 \times 3 \times k_2$
- Feature Maps: $(H - k_1 + 1) \times (W - k_2 + 1) \times 4$
Number of parameters with $N$ filters is: $N \times (k_1 \times k_2 \times 3 + 1)$

Example:
\[k_1 = 3, k_2 = 3, \text{ input channels } = 3, \text{ then } (3 \times 3 \times 3 + 1) \times 4 = 112\]
Just as before, in practice we can **vectorize** this operation.

**Step 1:** Lay out image patches in vector form (note can overlap!)

Adapted from: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/
Just as before, in practice we can **vectorize** this operation.

**Step 2:** Multiple patches by kernels

Input Matrix

![Input Matrix diagram]

<table>
<thead>
<tr>
<th>k</th>
<th>Patch 1</th>
<th>Patch 2</th>
<th>...</th>
</tr>
</thead>
</table>

Kernel Matrix

![Kernel Matrix diagram]

<table>
<thead>
<tr>
<th>Number of Kernels</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>Kernel 2</td>
<td></td>
</tr>
</tbody>
</table>

Adapted from: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/
Backwards Pass for Convolution Layer
It is instructive to calculate the **backwards pass** of a convolution layer

Similar to fully connected layer, will be **simple vectorized linear algebra operation**!

We will see a **duality** between cross-correlation and convolution

\[
K = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

\[
K' = \begin{bmatrix}
9 & 8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1
\end{bmatrix}
\]
Recap: Cross-Correlation

$$y(r, c) = (x * k)(r, c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r + a, c + b) k(a, b)$$
\[ y(r, c) = (x \ast k)(r, c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r + a, c + b) k(a, b) \]

Some simplification: 1 channel input, 1 kernel (channel output), padding (here 2 pixels on right/bottom) to make output the same size
\begin{align*}
y(r, c) &= (x \ast k)(r, c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r + a, c + b) k(a, b) \\
|y| &= H \times W
\end{align*}

\frac{\partial L}{\partial y} \quad \text{Assume size } H \times W \text{ (add padding, change convention a bit for convenience)}

\frac{\partial L}{\partial y(r, c)} \quad \text{to access element}
\[
\frac{\partial L}{\partial h_{\ell-1}} = \frac{\partial L}{\partial h_\ell} \frac{\partial h_\ell}{\partial h_{\ell-1}}
\]

Gradient for passing back

\[
\frac{\partial L}{\partial k} = \frac{\partial L}{\partial h_\ell} \frac{\partial h_\ell}{\partial k}
\]

Gradient for weight update

(\text{weights} = k, \text{i.e. kernel values})
Gradient for Convolution Layer
\[
\frac{\partial L}{\partial k} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial k}
\]

Gradient for weight update

Calculate one pixel at a time

\[
\frac{\partial L}{\partial k(a,b)}
\]

What does this weight affect at the output?

Everything!

What a Kernel Pixel Affects at Output
Need to incorporate all upstream gradients:
\[
\left\{ \frac{\partial L}{\partial y(0, 0)}, \frac{\partial L}{\partial y(0, 1)}, \ldots, \frac{\partial L}{\partial y(H, W)} \right\}
\]

Chain Rule:
\[
\frac{\partial L}{\partial k(a, b)} = \sum_{r=0}^{H-1} \sum_{c=0}^{W-1} \frac{\partial L}{\partial y(r, c)} \frac{\partial y(r, c)}{\partial k(a, b)}
\]

Sum over all output pixels
Upstream gradient (known)
We will compute

\((0, 0)\)

\(H = 5\)

\(W = 5\)

\((H - 1, W - 1)\)
Chain Rule over all Output Pixels
\[ \frac{\partial y(r, c)}{\partial k(a, b)} = x(r + a, c + b) \]

\[ \frac{\partial L}{\partial k(a, b)} = \sum_{r=0}^{H-1} \sum_{c=0}^{W-1} \frac{\partial L}{\partial y(r, c)} x(r + a, c + b) \]

Does this look familiar?

Cross-correlation between upstream gradient and input!
(until \( k_1 \times k_2 \) output)
Forward and Backward Duality

Does this look familiar?

Cross-correlation between upstream gradient and input!
(unttul $k_1 \times k_2$ output)

Forward Pass

Backward Pass $k(0, 0)$

Backward Pass $k(2, 2)$

$\frac{\partial L}{\partial y}$
Gradient for input (to pass to prior layer)

Calculate one pixel at a time \( \frac{\partial L}{\partial x(r',c')} \)

What does this input pixel affect at the output?

Neighborhood around it (where part of the kernel touches it)

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

(0, 0)

\( H = 5 \)

\( W = 5 \)

\( (0, 0) \)

\( r',c' \)

\( k_1 = 3 \)

\( k_2 = 3 \)

\( (k_1 - 1, k_2 - 1) \)
Extents of Kernel Touching the Pixel
This is where the corresponding locations are for the output \((r' - k_1 + 1, \ c' - k_2 + 1)\).
Chain rule for affected pixels (sum gradients):

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}
\]

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(? , ?)} \frac{\partial y(?, ?)}{\partial x(r', c')}
\]

1. \(H = 5\)
2. \(W = 5\)

(r' - k_1 + 1, c' - k_2 + 1)
Summing Gradient Contributions

Let's derive it analytically this time (as opposed to visually)

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}
\]

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} \frac{\partial y(r' - a, c' - b)}{\partial x(r', c')}
\]

\[(r' - k_1 + 1, c' - k_2 + 1)\]
Definition of cross-correlation (use \( a', b' \) to distinguish from prior variables):

\[
y(r', c') = (x \ast k)(r', c') = \sum_{a'=0}^{k_1-1} \sum_{b'=0}^{k_2-1} x(r' + a', c' + b') k(a', b')
\]

Plug in what we actually wanted:

\[
y(r' - a, c' - b) = (x \ast k)(r', c') = \sum_{a'=0}^{k_1-1} \sum_{b'=0}^{k_2-1} x(r' - a + a', c' - b + b') k(a', b')
\]

What is \( \frac{\partial y(r' - a, c' - b)}{\partial x(r', c')} = k(a, b) \) (we want term with \( x(r', c') \) in it; this happens when \( a = a' \) and \( b = b' \))
Plugging in to earlier equation:

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} \frac{\partial y(r' - a, c' - b)}{\partial x(r', c')}
\]

\[
= \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} k(a, b)
\]

Does this look familiar?

Convolution between upstream gradient and kernel!
(can implement by flipping kernel and cross-correlation)

Again, all operations can be implemented via matrix multiplications (same as FC layer)!
• Convolutions are mathematical descriptions of striding linear operation

• In practice, we implement **cross-correlation neural networks!** (still called convolutional neural networks due to history)
  • Can connect to convolutions via duality (flipping kernel)
  • Convolution formulation has mathematical properties explored in ECE

• Duality for forwards and backwards:
  • Forward: Cross-correlation
  • Backwards w.r.t. K: Cross-correlation b/w upstream gradient and input
  • Backwards w.r.t. X: Convolution b/w upstream gradient and kernel
    • In practice implement via cross-correlation and flipped kernel

• All operations still implemented via efficient linear algebra (e.g. matrix-matrix multiplication)
Topics:
• Convolutional Neural Networks
Mathematics of Discrete 2D Convolution

\[ y(r, c) = (x * k)(r, c) = \sum_{a=-\frac{H-1}{2}}^{\frac{H-1}{2}} \sum_{b=-\frac{W-1}{2}}^{\frac{W-1}{2}} x(a, b) k(r - a, c - b) \]

\[ \left( \begin{array}{cc} -\frac{H-1}{2} & -\frac{W-1}{2} \end{array} \right) \]

\[ H = 5 \]

\[ W = 5 \]

\[ \left( \begin{array}{cc} \frac{H-1}{2} & \frac{W-1}{2} \end{array} \right) \]

\[ k_1 = 3 \]

\[ k_2 = 3 \]

\[ y(0, 0) = x(-2, -2)k(2, 2) + x(-2, -1)k(2, 1) + x(-2, 0)k(2, 0) + x(-2, 1)k(2, -1) + x(-2, 2)k(2, -2) + \ldots \]
\[ y(r, c) = (x \ast k)(r, c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r + a, c + b) k(a, b) \]

Since we will be learning these kernels, this change does not matter!
\[x(0:2,0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix} \quad K' = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \]

\[x(0:2,0:2) \cdot K' = 65 + \text{bias} \]

Dot product
(element-wise multiply and sum)
Convolution and Cross-Correlation
Convolution and Cross-Correlation
Convolution and Cross-Correlation
Convolution and Cross-Correlation
Convolution and Cross-Correlation
Why Bother with Convolutions?

Convolutions are just simple linear operations

Why bother with this and not just say it’s a linear layer with small receptive field?

- There is a duality between them during backpropagation
- Convolutions have various mathematical properties people care about
- This is historically how it was inspired
We can **pad the images** to make the output the same size:

- Zeros, mirrored image, etc.
- Note padding often refers to pixels added to **one size** ($P = 1$ here)

\[
\begin{align*}
H + 2 & \quad W + 2 \\
0 & \quad \ldots \\
\ldots & \\
\end{align*}
\]

\[
\begin{align*}
H + 2 - k_1 + 1 & \\
W + 2 - k_2 + 1 &
\end{align*}
\]
We can have **multiple kernels per layer**

- We stack the feature maps together at the output

Number of channels in output is equal to *number of kernels*
Number of parameters with $N$ filters is: $N \times (k_1 \times k_2 \times 3 + 1)$

Example:
$k_1 = 3, k_2 = 3, \ N = 4$ input channels $= 3$, then $(3 \times 3 \times 3 + 1) \times 4 = 112$
Need to incorporate all upstream gradients:

\[
\left\{ \frac{\partial L}{\partial y(0, 0)} \cdot \frac{\partial L}{\partial y(0, 1)} \cdots \frac{\partial L}{\partial y(H, W)} \right\}
\]

Chain Rule:

\[
\frac{\partial L}{\partial k(a, b)} = \sum_{r=0}^{H-1} \sum_{c=0}^{W-1} \frac{\partial L}{\partial y(r, c)} \frac{\partial y(r, c)}{\partial k(a, b)}
\]

Upstream gradient (known)  
Sum over all output pixels  
We will compute

\((0, 0)\)

\((0, 0)\)

\((H - 1, W - 1)\)

\(H = 5\)

\(W = 5\)

\(k_1 = 3\)

\(k_2 = 3\)

\((k_1 - 1, k_2 - 1)\)
\[
\frac{\partial y(r, c)}{\partial k(a, b)} = ?
\]

**Reasoning:**
- Cross-correlation is just “dot product” of kernel and input patch (weighted sum)
- When at pixel \(y(r, c)\), kernel is on input \(x\) such that \(k(0, 0)\) is multiplied by \(x(r, c)\)
- But we want derivative \(\text{w.r.t.} \ k(a, b)\)
  - \(k(0, 0) \times x(r, c), k(1, 1) \times x(r + 1, c + 1), k(2, 2) \times x(r + 2, c + 2) \Rightarrow \text{in general} \ k(a, b) \times x(r + a, c + b)\)
  - Just like before in fully connected layer, partial derivative \(\text{w.r.t.} \ k(a, b)\) only has this term (other \(x\) terms go away because not multiplied by \(k(a, b)\)).
\[
\frac{\partial y(r, c)}{\partial k(a, b)} = x(r + a, c + b)
\]

\[
\frac{\partial L}{\partial k(a, b)} = \sum_{r=0}^{H-1} \sum_{c=0}^{W-1} \frac{\partial L}{\partial y(r, c)} x(r + a, c + b)
\]

Does this look familiar?

Cross-correlation between upstream gradient and input!
(until \(k_1 \times k_2\) output)
Cross-correlation between upstream gradient and input!
(until $k_1 \times k_2$ output)
\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x}
\]

Gradient for input (to pass to prior layer)

Calculate one pixel at a time \( \frac{\partial L}{\partial x(r',c')} \)

What does this input pixel affect at the output?

Neighborhood around it (where part of the kernel touches it)

\( W = 5 \)
\( H = 5 \)

\( (0,0) \)

\( k_1 = 3 \)

\( k_2 = 3 \)

\( (k_1 - 1, k_2 - 1) \)
Extents of Kernel Touching the Pixel
This is where the corresponding locations are for the output.

Extents at the Output

\[ k_1 = 3 \]
\[ k_2 = 3 \]

\[ W = 5 \]
\[ H = 5 \]

\[ (r' - k_1 + 1, c' - k_2 + 1) \]
Chain rule for affected pixels (sum gradients):

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}
\]

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(?, ?)} \frac{\partial y(?, ?)}{\partial x(r', c')}
\]

\[
x(r', c') \cdot k(0, 0) \Rightarrow y(r', c') \\
x(r', c') \cdot k(1, 1) \Rightarrow ?
\]

\[
(r' - k_1 + 1, c' - k_2 + 1)
\]
Chain rule for affected pixels (sum gradients):

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}
\]

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(? , ?)} \frac{\partial y(? , ?)}{\partial x(r', c')}
\]

\[x(r', c') \ast k(0, 0) \Rightarrow y(r', c')\]
\[x(r', c') \ast k(1, 1) \Rightarrow y(r' - 1, c' - 1)\]
\[\ldots\]
\[x(r', c') \ast k(a, b) \Rightarrow y(r' - a, c' - b)\]
Chain rule for affected pixels (sum gradients):

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}
\]

Let's derive it analytically this time (as opposed to visually)

\[
\begin{align*}
\frac{\partial L}{\partial x(r', c')} &= \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} \frac{\partial y(r' - a, c' - b)}{\partial x(r', c')} \\
&= (r' - k_1 + 1, c' - k_2 + 1)
\end{align*}
\]
Definition of cross-correlation (use $a', b'$ to distinguish from prior variables):

\[
y(r', c') = (x \ast k)(r', c') = \sum_{a'=0}^{k_1-1} \sum_{b'=0}^{k_2-1} x(r' + a', c' + b') \ k(a', b')
\]

Plug in what we actually wanted:

\[
y(r' - a, c' - b) = (x \ast k)(r', c') = \sum_{a'=0}^{k_1-1} \sum_{b'=0}^{k_2-1} x(r' - a + a', c' - b + b') \ k(a', b')
\]

What is \(\frac{\partial y(r' - a, c' - b)}{\partial x(r', c')} = k(a, b)\) (we want term with \(x(r', c')\) in it; this happens when \(a = a'\) and \(b = b'\))
Plugging in to earlier equation:

\[
\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} \frac{\partial y(r' - a, c' - b)}{\partial x(r', c')}
\]

\[
= \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} k(a, b)
\]

Does this look familiar?

Convolution between upstream gradient and kernel!
(can implement by flipping kernel and cross-correlation)

Again, all operations can be implemented via matrix multiplications (same as FC layer)!
• Convolutions are mathematical descriptions of striding linear operation

• In practice, we implement cross-correlation neural networks! (still called convolutional neural networks due to history)
  • Can connect to convolutions via duality (flipping kernel)
  • Convolution formulation has mathematical properties explored in ECE

• Duality for forwards and backwards:
  • **Forward**: Cross-correlation
  • **Backwards w.r.t. K**: Cross-correlation b/w upstream gradient and input
  • **Backwards w.r.t. X**: Convolution b/w upstream gradient and kernel
    • In practice implement via cross-correlation and flipped kernel

• All operations still implemented via **efficient linear algebra** (e.g. matrix-matrix multiplication)
Pooling Layers
Dimensionality reduction is an important aspect of machine learning.

Can we make a layer to **explicitly down-sample** image or feature maps?

**Yes!** We call one class of these operations **pooling** operations.

---

**Parameters**

- `kernel_size` – the size of the window to take a max over
- `stride` – the stride of the window. Default value is `kernel_size`
- `padding` – implicit zero padding to be added on both sides

Example: Max pooling

- Stride window across image but perform per-patch max operation

\[
X(0:2, 0:2) = \begin{bmatrix}
200 & 150 & 150 \\
100 & 50 & 100 \\
25 & 25 & 10
\end{bmatrix} \Rightarrow \text{max}(0:2,0:2) = 200
\]

How many learned parameters does this layer have?
None!
Not restricted to max; can use any differentiable function

- Not very common in practice

\[
X(0:2, 0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix}
\]

\[
\text{average}(0:2,0:2) = \frac{1}{N} \sum_i \sum_j x(i,j) = 90
\]
Since the **output** of convolution and pooling layers are *(multi-channel) images*, we can sequence them just as any other layer.

\[
\begin{align*}
H &= 5 \\
W &= 5
\end{align*}
\]
This combination adds some **invariance** to translation of the features.

- If feature (such as beak) translated a little bit, output values still remain the same.

\[ W = 5 \]
\[ H = 5 \]

- Image
  - Convolution Layer
  - Pooling Layer

**Invariance**
Convolution by itself has the property of **equivariance**

- If feature (such as beak) translated a little bit, output values **move by the same translation**

\[
\mathbf{W} = 5 \\
\mathbf{H} = 5
\]
Since the **output** of convolution and pooling layers are *(multi-channel) images*, we can sequence them just as any other layer.

\[ \text{Convolution Layer} \]
\[ \text{Pooling Layer} \]

\( W = 5 \)
\( H = 5 \)

Image
Convolutional Neural Networks (CNNs)

Image

Convolution + Non-Linear Layer

Pooling Layer

Convolution + Non-Linear Layer

Useful, lower-dimensional features

Alternating Convolution and Pooling
Adding a Fully Connected Layer

1. Image
2. Convolution + Non-Linear Layer
3. Pooling Layer
4. Convolution + Non-Linear Layer
5. Fully Connected Layers

Loss
Receptive Fields

- Image
- Convolution + Non-Linear Layer
- Pooling Layer
- Convolution + Non-Linear Layer
- Fully Connected Layers
- Loss
Typical Depiction of CNNs
These architectures have existed since 1980s

LeNet Architecture

INPUT 32x32

Convolutions

C1: feature maps 6@28x28

Subsampling

S2: f. maps 6@14x14

Convolutions

C3: f. maps 16@10x10

Subsampling

S4: f. maps 16@5x5

Convolutions

C5: layer 120

Subsampling

F6: layer 84

Full connection

Gaussian connections

OUTPUT 10

Image Credit: Yann LeCun, Kevin Murphy
Handwriting Recognition

Image Credit: Yann LeCun
Translation Equivariance (Conv Layers) & Invariance (Output)
(Some) Rotation Invariance
(Some) Scale Invariance
Advanced
Convolutional
Networks
The Importance of Benchmarks

From: https://paperswithcode.com
AlexNet - Architecture

From: Krizhevsky et al., ImageNet Classification with Deep Convolutional Neural Networks, 2012.
Key aspects:
- ReLU instead of sigmoid or tanh
- Specialized normalization layers
- PCA-based data augmentation
- Dropout
- Ensembling

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)
From: Simonyan & Zimmerman, Very Deep Convolutional Networks for Large-Scale Image Recognition
From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Most memory usage in convolution layers

Most parameters in FC layers

From: Simonyan & Zimmerman, Very Deep Convolutional Networks for Large-Scale Image Recognition
From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Key aspects:

Repeated application of:

- 3x3 conv (stride of 1, padding of 1)
- 2x2 max pooling (stride 2)

Very large number of parameters

From: Simonyan & Zimmerman, Very Deep Convolutional Networks for Large-Scale Image Recognition
From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
But have become **deeper and more complex**

*From: Szegedy et al. Going deeper with convolutions*
Key idea: Repeated blocks and multi-scale features

From: Szegedy et al. Going deeper with convolutions
The Challenge of Depth

Optimizing very deep networks is challenging!

From: He et al., Deep Residual Learning for Image Recognition
Residual Blocks and Skip Connections

**Key idea**: Allow information from a layer to propagate to any future layer (forward)

Same is true for gradients!

*From: He et al., Deep Residual Learning for Image Recognition*
Several ways to learn architectures:

- Evolutionary learning and reinforcement learning
- Prune over-parameterized networks
- Learning of repeated blocks typical

From: https://ai.googleblog.com/2018/03/using-evolutionary-automl-to-discover.html
Computational Complexity

From: An Analysis Of Deep Neural Network Models For Practical Applications