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

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

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

Some simplification: 1 channel input, 1 kernel (channel output), padding (here 2 pixels on right/bottom) to make output the same size
Gradient Terms and Notation

Assume size \((x \ast k)(r, c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r + a, c + b) k(a, b)\)

\(|y| = H \times W\)

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

\(\frac{\partial L}{\partial y(r, c)}\) to access element
Backpropagation Chain Rule

\[ \frac{\partial L}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial h^{\ell-1}} \]

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

(weights = k, i.e. kernel values)
Gradient for Convolution Layer
What a Kernel Pixel Affects at Output

\[
\frac{\partial L}{\partial k} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial k}
\]

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!

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

\( k_1 = 3 \)
\( k_2 = 3 \)

(0, 0) \( \rightarrow (0, 0) \)

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

(\( k_1 - 1 \), \( k_2 - 1 \))
Need to incorporate all upstream gradients:
\[
\begin{align*}
\left\{ \frac{\partial L}{\partial y(0, 0)} \frac{\partial L}{\partial y(0, 1)} \cdots \frac{\partial L}{\partial y(H, W)} \right\}
\end{align*}
\]

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

\(k_1 = 3\)
\(k_2 = 3\)

\((0, 0)\) \(\rightarrow\) \((0, 0)\)

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

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

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

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

Does this look familiar?
Forward and Backward Duality

Does this look familiar?

Cross-correlation between upstream gradient and input!

(until $k_1 \times k_2$ output)

Forward Pass

Backward Pass $k(0, 0)$

Backward Pass $k(2, 2)$

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

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)

\( W = 5 \)
\( H = 5 \)
\( k_1 = 3 \)
\( k_2 = 3 \)

\((0, 0)\) \( (0, 0) \)
\((H - 1, W - 1)\) \( (k_1 - 1, k_2 - 1) \)
Extents of Kernel Touching the Pixel

1. 
2. 
3. 
4.
Extents at the Output

This is where the corresponding locations are for the output

\( \text{Extents at the Output} \)

\[ k_1 = 3 \]

\[ k_2 = 3 \]

\( (r' - k_1 + 1, c' - k_2 + 1) \)

\( W = 5 \)

\( H = 5 \)
Let’s derive it analytically this time (as opposed to visually).

Chain rule for affected pixels (sum gradients):

$$\frac{\partial L}{\partial x(r', c')} = \sum_{\text{pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}$$

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

$$k_1 = 3$$
$$k_2 = 3$$

Summing Gradient Contributions
Calculating the Gradient

Definition of cross-correlation (use $a', b'$ to distinguish from prior variables):

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

Plug in what we actually wanted:

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

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)!
Simple Convolutional Neural Networks
Since the **output** of convolution and pooling layers are *(multi-channel) images*, we can sequence them just as any other layer.
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

Image → Convolution + Non-Linear Layer → Pooling Layer → Convolution + Non-Linear Layer → Fully Connected Layers → Loss
Receptive Fields
Convolutional Neural Networks

Input Image → Predictions
These architectures have existed since 1980s

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

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

Image Credit: Yann LeCun
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.
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
[13x13x384] 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)

Key aspects:
- ReLU instead of sigmoid or tanh
- Specialized normalization layers
- PCA-based data augmentation
- Dropout
- Ensembling

From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
**INPUT:** [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)

**CONV3-64:** [224x224x64] memory: 224*224*64=3.2M params: (3*3)*64 = 1,728

**CONV3-64:** [224x224x64] memory: 224*224*64=3.2M params: (3*3)*64 = 36,864

**POOL2:** [112x112x64] memory: 112*112*64=800K params: 0

**CONV3-128:** [112x112x128] memory: 112*112*128=1.6M params: (3*3)*128 = 73,728

**CONV3-128:** [112x112x128] memory: 112*112*128=1.6M params: (3*3)*128 = 147,456

**POOL2:** [56x56x128] memory: 56*56*128=400K params: 0

**CONV3-256:** [56x56x256] memory: 56*56*256=800K params: (3*3)*256 = 294,912

**CONV3-256:** [56x56x256] memory: 56*56*256=800K params: (3*3)*256 = 589,824

**CONV3-256:** [56x56x256] memory: 56*56*256=800K params: (3*3)*256 = 589,824

**POOL2:** [28x28x256] memory: 28*28*256=200K params: 0

**CONV3-512:** [28x28x512] memory: 28*28*512=400K params: (3*3)*512 = 1,179,648

**CONV3-512:** [28x28x512] memory: 28*28*512=400K params: (3*3)*512 = 2,359,296

**CONV3-512:** [28x28x512] memory: 28*28*512=400K params: (3*3)*512 = 2,359,296

**POOL2:** [14x14x512] memory: 14*14*512=100K params: 0

**CONV3-512:** [14x14x512] memory: 14*14*512=100K params: (3*3)*512 = 2,359,296

**CONV3-512:** [14x14x512] memory: 14*14*512=100K params: (3*3)*512 = 2,359,296

**CONV3-512:** [14x14x512] memory: 14*14*512=100K params: (3*3)*512 = 2,359,296

**POOL2:** [7x7x512] memory: 7*7*512=25K params: 0

**FC:** [1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448

**FC:** [1x4096] memory: 4096 params: 4096*4096 = 16,777,216

**FC:** [1x1000] memory: 1000 params: 4096*1000 = 4,096,000

**Table 2: Number of parameters** (in millions).

<table>
<thead>
<tr>
<th>Network</th>
<th>A</th>
<th>A-LRN</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of parameters</td>
<td>133</td>
<td>133</td>
<td>134</td>
<td>138</td>
<td>144</td>
<td></td>
</tr>
</tbody>
</table>

*From: Simonyan & Zimmerman, Very Deep Convolutional Networks for Large-Scale Image Recognition*

*From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*
### Parameters and Memory

<table>
<thead>
<tr>
<th>Layer</th>
<th>Dimension</th>
<th>Memory</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT: [224x224x3]</td>
<td>memory: 224<em>224</em>3=150K</td>
<td>(not counting biases)</td>
<td>params: 0</td>
</tr>
<tr>
<td>CONV3-64: [224x224x64]</td>
<td>memory: 224<em>224</em>64=3.2M</td>
<td>params: (3<em>3</em>64) = 1,728</td>
<td></td>
</tr>
<tr>
<td>CONV3-64: [224x224x64]</td>
<td>memory: 224<em>224</em>64=3.2M</td>
<td>params: (3<em>3</em>64) = 1,728</td>
<td></td>
</tr>
<tr>
<td>POOL2: [112x112x64]</td>
<td>memory: 112<em>112</em>64=800K</td>
<td>params: 0</td>
<td></td>
</tr>
<tr>
<td>CONV3-128: [112x112x128]</td>
<td>memory: 112<em>112</em>128=1.6M</td>
<td>params: (3<em>3</em>128)*128 = 73,728</td>
<td></td>
</tr>
<tr>
<td>CONV3-128: [112x112x128]</td>
<td>memory: 112<em>112</em>128=1.6M</td>
<td>params: (3<em>3</em>128)*128 = 73,728</td>
<td></td>
</tr>
<tr>
<td>POOL2: [56x56x128]</td>
<td>memory: 56<em>56</em>128=400K</td>
<td>params: 0</td>
<td></td>
</tr>
<tr>
<td>CONV3-256: [56x56x256]</td>
<td>memory: 56<em>56</em>256=800K</td>
<td>params: (3<em>3</em>256)*256 = 294,912</td>
<td></td>
</tr>
<tr>
<td>CONV3-256: [56x56x256]</td>
<td>memory: 56<em>56</em>256=800K</td>
<td>params: (3<em>3</em>256)*256 = 294,912</td>
<td></td>
</tr>
<tr>
<td>CONV3-256: [56x56x256]</td>
<td>memory: 56<em>56</em>256=800K</td>
<td>params: (3<em>3</em>256)*256 = 294,912</td>
<td></td>
</tr>
<tr>
<td>POOL2: [28x28x256]</td>
<td>memory: 28<em>28</em>256=200K</td>
<td>params: 0</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: [28x28x512]</td>
<td>memory: 28<em>28</em>512=400K</td>
<td>params: (3<em>3</em>512)*512 = 1,179,648</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: [28x28x512]</td>
<td>memory: 28<em>28</em>512=400K</td>
<td>params: (3<em>3</em>512)*512 = 1,179,648</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: [28x28x512]</td>
<td>memory: 28<em>28</em>512=400K</td>
<td>params: (3<em>3</em>512)*512 = 1,179,648</td>
<td></td>
</tr>
<tr>
<td>POOL2: [14x14x512]</td>
<td>memory: 14<em>14</em>512=100K</td>
<td>params: 0</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: [14x14x512]</td>
<td>memory: 14<em>14</em>512=100K</td>
<td>params: (3<em>3</em>512)*512 = 2,359,296</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: [14x14x512]</td>
<td>memory: 14<em>14</em>512=100K</td>
<td>params: (3<em>3</em>512)*512 = 2,359,296</td>
<td></td>
</tr>
<tr>
<td>CONV3-512: [14x14x512]</td>
<td>memory: 14<em>14</em>512=100K</td>
<td>params: (3<em>3</em>512)*512 = 2,359,296</td>
<td></td>
</tr>
<tr>
<td>POOL2: [7x7x512]</td>
<td>memory: 7<em>7</em>512=25K</td>
<td>params: 0</td>
<td></td>
</tr>
<tr>
<td>FC: [1x4096]</td>
<td>memory: 4096</td>
<td>params: 7<em>7</em>512*4096 = 102,760,448</td>
<td></td>
</tr>
<tr>
<td>FC: [1x4096]</td>
<td>memory: 4096</td>
<td>params: 4096*4096 = 16,777,216</td>
<td></td>
</tr>
<tr>
<td>FC: [1x1000]</td>
<td>memory: 1000</td>
<td>params: 4096*1000 = 4,096,000</td>
<td></td>
</tr>
</tbody>
</table>

From: Simonyan & Zimmermann, 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 & Zimmermann, 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**

**Inception Architecture**

*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
Transfer Learning & Generalization
Multi-class Logistic Regression

- Softmax
- FC HxWx3
- Input

Model class

Optimization Error = 0

Estimation Error

Modeling Error

Reality

From: slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Generalization model class

From: slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
What if we don’t have enough data?

**Step 1:** Train on large-scale dataset

![Convolutional Neural Networks](image)

**Input Image** → **Predictions**
Step 2: Take your custom data and **initialize** the network with weights trained in Step 1

Replace last layer with new fully-connected for output nodes per new category
Step 3: (Continue to) train on new dataset

- **Finetune**: Update all parameters
- **Freeze** feature layer: Update only last layer weights (used when not enough data)

Replace last layer with new fully-connected for output nodes per new category
This works extremely well! It was surprising upon discovery.

- Features learned for 1000 object categories will work well for 1001st!

- Generalizes even across tasks (classification to object detection)

From: Razavian et al., CNN Features off-the-shelf: an Astounding Baseline for Recognition
But it doesn’t always work that well!

- If the **source** dataset you train on is very different from the **target** dataset, transfer learning is not as effective.

- If you have enough data for the target domain, it just results in faster convergence.

  See He et al., “Rethinking ImageNet Pre-training”
Effectiveness of More Data

From: Revisiting the Unreasonable Effectiveness of Data

From: Hestness et al., Deep Learning Scaling Is Predictable

Figure 6: Sketch of power-law learning curves
There is a large number of different low-labeled settings in DL research

<table>
<thead>
<tr>
<th>Setting</th>
<th>Source</th>
<th>Target</th>
<th>Shift Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-supervised</td>
<td>Single labeled</td>
<td>Single unlabeled</td>
<td>None</td>
</tr>
<tr>
<td>Domain Adaptation</td>
<td>Single labeled</td>
<td>Single unlabeled</td>
<td>Non-semantic</td>
</tr>
<tr>
<td>Domain Generalization</td>
<td>Multiple labeled</td>
<td>Unknown</td>
<td>Non-semantic</td>
</tr>
<tr>
<td>Cross-Category Transfer</td>
<td>Single labeled</td>
<td>Single unlabeled</td>
<td>Semantic</td>
</tr>
<tr>
<td>Few-Shot Learning</td>
<td>Single labeled</td>
<td>Single few-labeled</td>
<td>Semantic</td>
</tr>
<tr>
<td>Un/Self-Supervised</td>
<td>Single unlabeled</td>
<td>Many labeled</td>
<td>Both/Task</td>
</tr>
</tbody>
</table>

Non-Semantic Shift

Semantic Shift