# Backwards Pass for Convolution Layer

# Backwards Pass for Conv Layers

It is instructive to calculate **the backwards pass** of a convolution layer

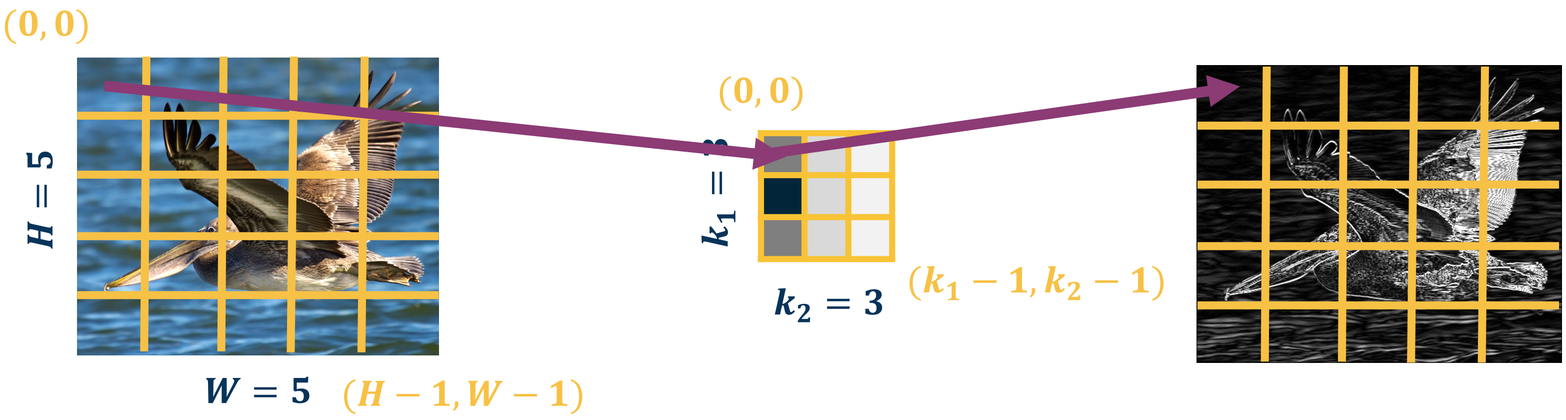
- Similar to fully connected layer, will be **simple vectorized linear algebra operation**!
- We will see a **duality** between cross-correlation and convolution

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

↓

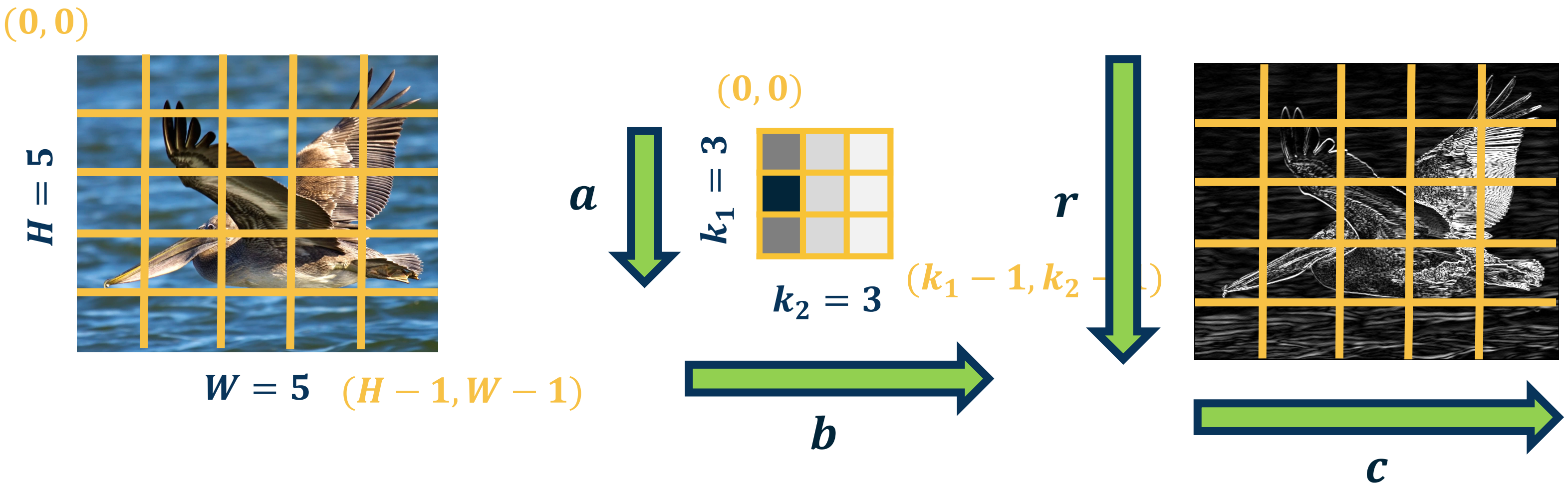
$$K' = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

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



# Recap: Cross-Correlation

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



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

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

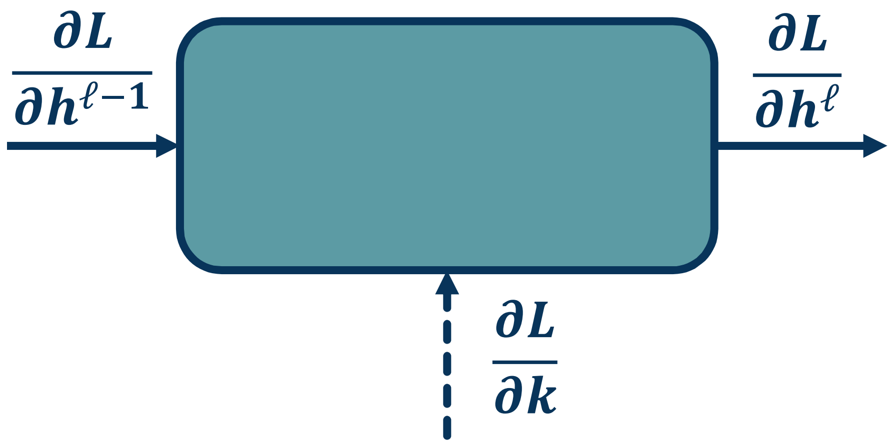
$$|\boldsymbol{y}| = \boldsymbol{H} \times \boldsymbol{W}$$

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

$\frac{\partial L}{\partial y(r,c)}$ to access element

# Gradient Terms and Notation

# Backpropagation Chain Rule



$\frac{\partial L}{\partial h^{\ell-1}}$

$\frac{\partial L}{\partial h^{\ell}}$

$\frac{\partial L}{\partial k}$

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

**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

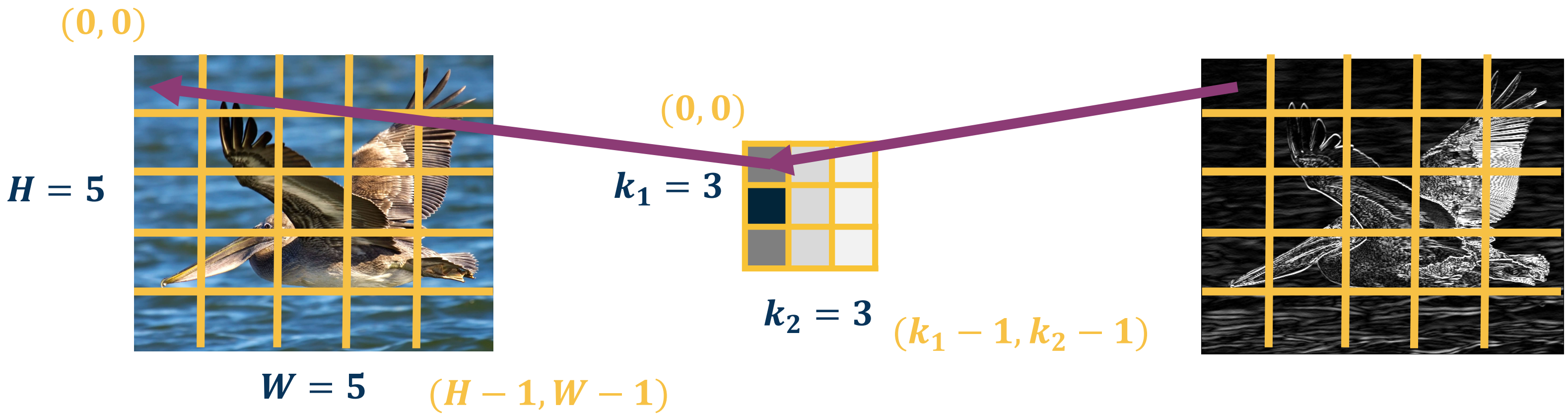
$$\frac{\partial L}{\partial k} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial k}$$

**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!**



# Chain Rule over all Output Pixels

Need to incorporate all upstream gradients:

$$\left\{\frac{\partial L}{\partial y(0,0)}, \frac{\partial L}{\partial y(0,1)}, \dots, \frac{\partial L}{\partial y(H,W)}\right\}$$

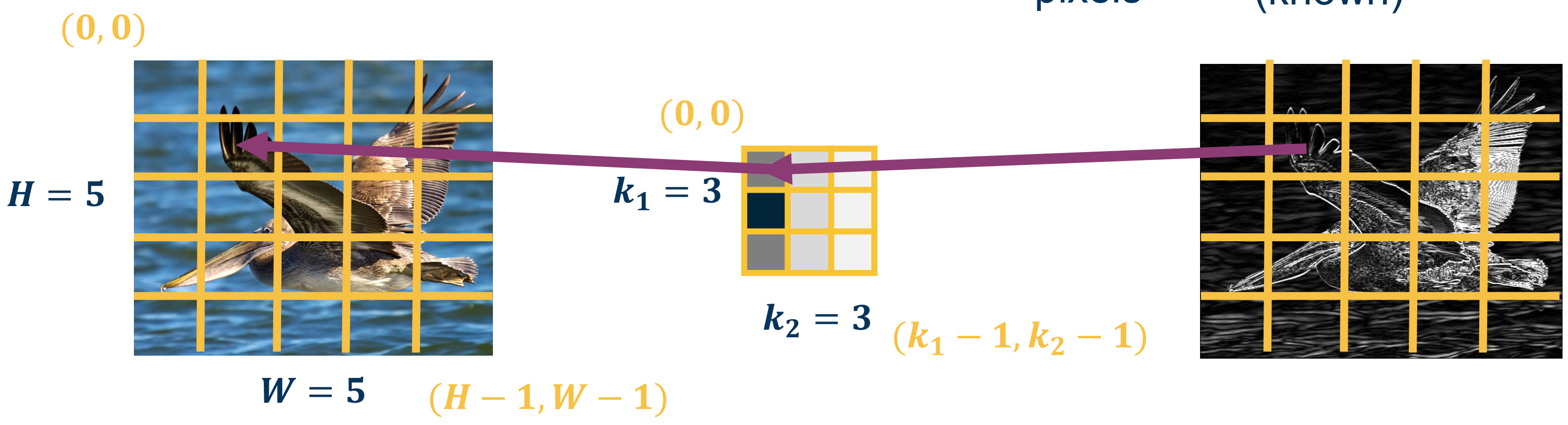
Chain Rule:

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

Sum over all output pixels

Upstream gradient (known)

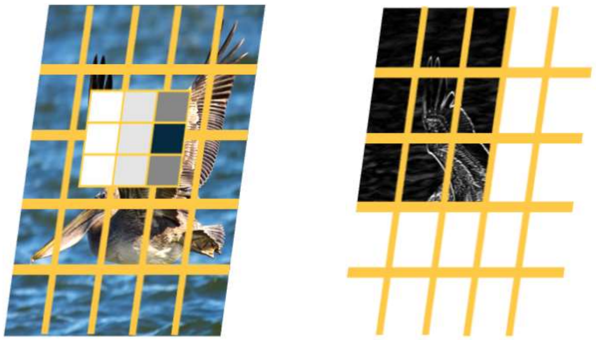
We will compute

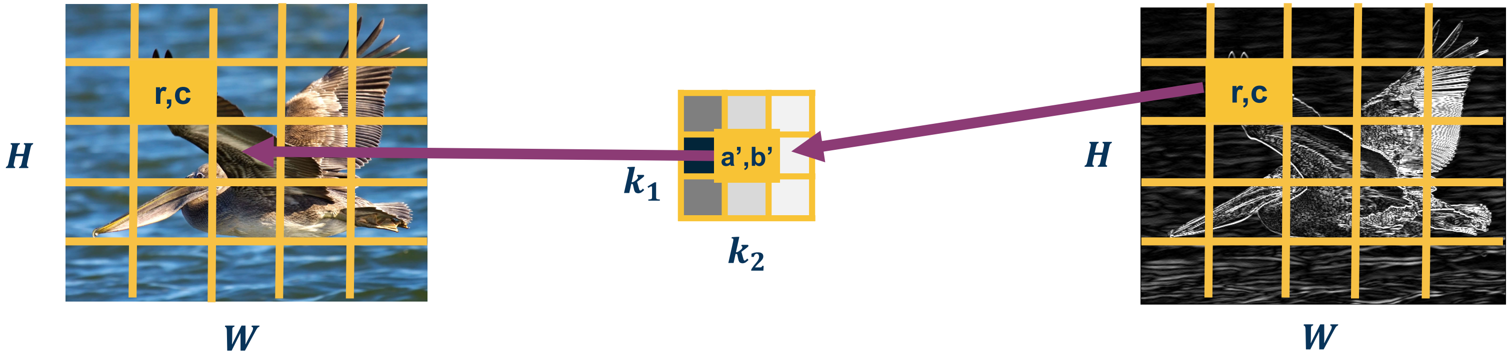


# Chain Rule over all Output Pixels

$$\frac{\partial y(r,c)}{\partial k(a',b')} = x(r+a',c+b')$$

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





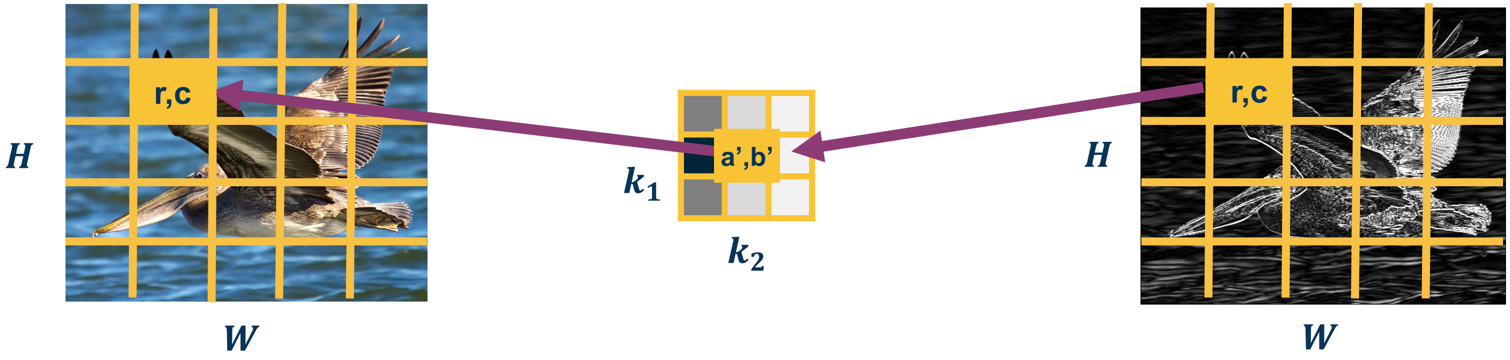
$$\frac{\partial y(r,c)}{\partial k(a',b')} = x(r+a', c+b')$$

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

Does this look familiar?

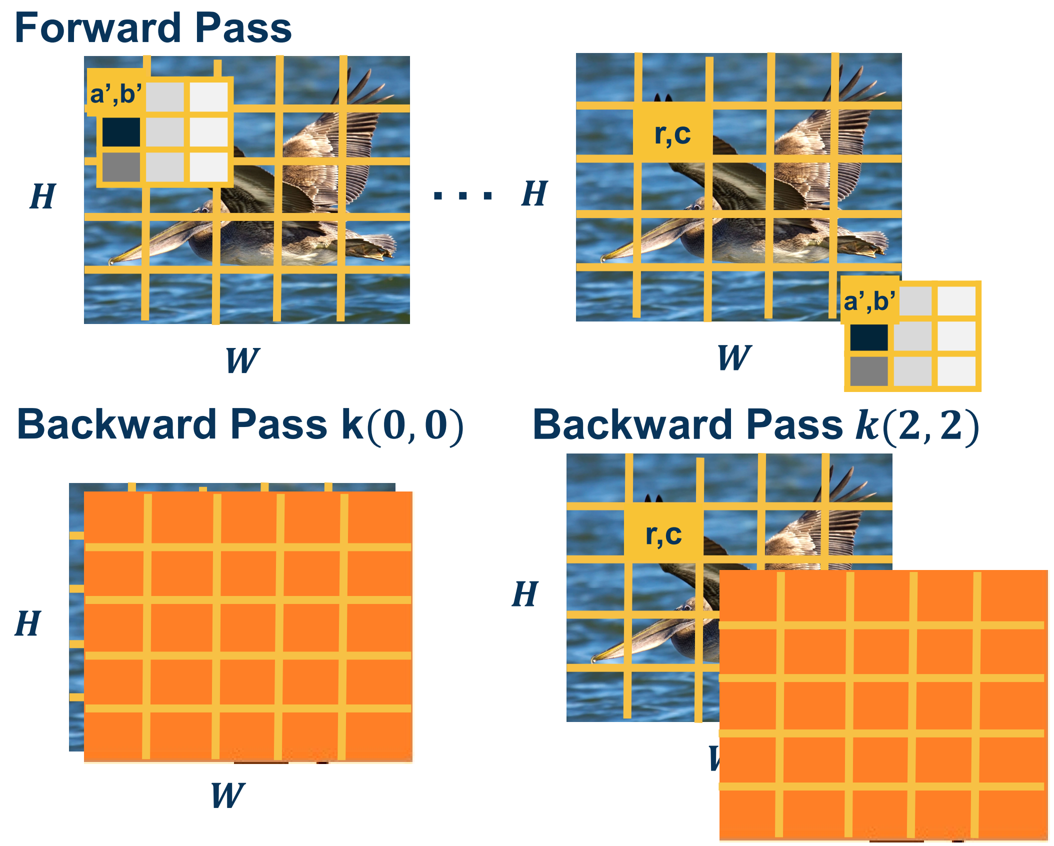
Cross-correlation between upstream gradient and input!

(until $k_1 \times k_2$ output)



# Gradients and Cross-Correlation

# Forward and Backward Duality



Forward Pass

Backward Pass $k(0,0)$

Backward Pass $k(2,2)$

**Does this look familiar?**

**Cross-correlation between upstream gradient and input!**

**(until $k_1 \times k_2$ output)**

## What an Input Pixel Affects at Output

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

Gradient for input (to pass to prior layer)

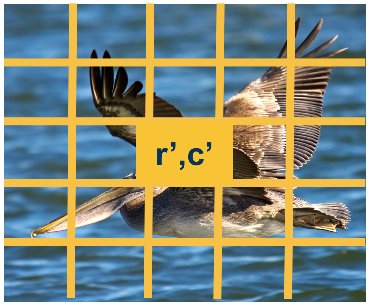
Calculate one pixel at a time $\frac{\partial L}{\partial x(r',c')}$

**What does this input pixel affect at the output?**

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

$(0,0)$

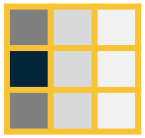
$H = 5$



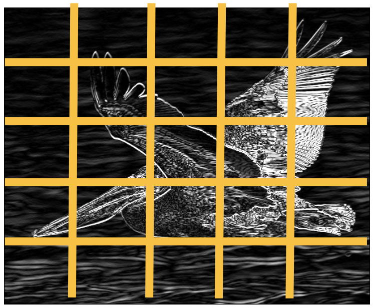
$W = 5$ $(H-1, W-1)$

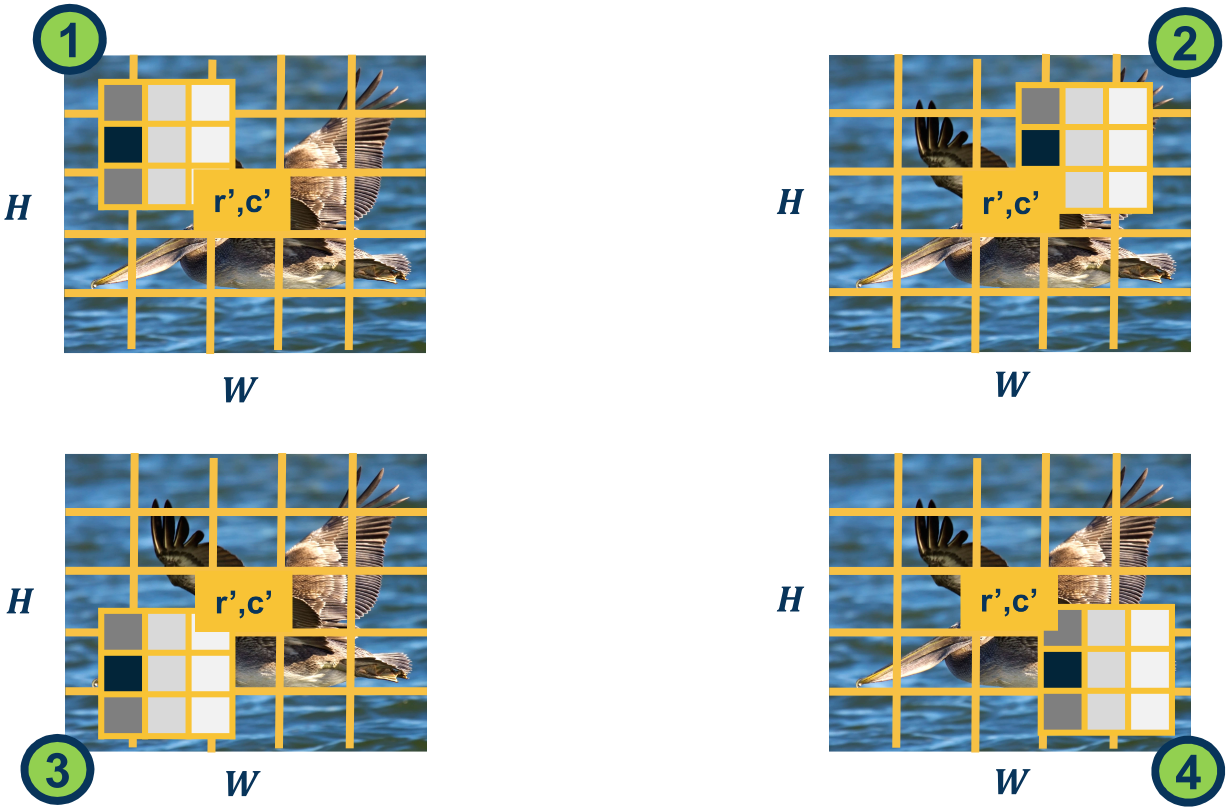
$(0,0)$

$k_1 = 3$



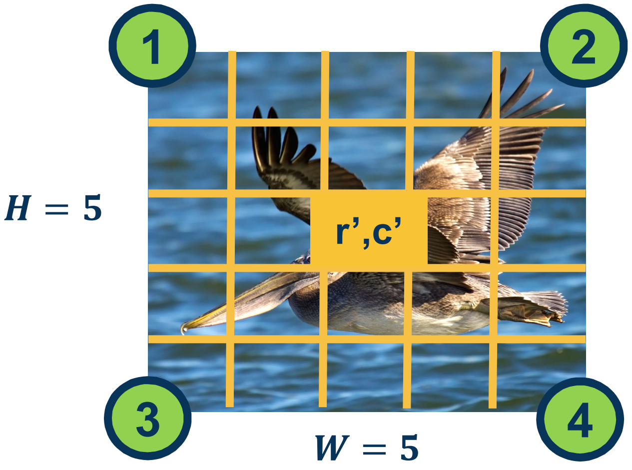
$k_2 = 3$ $(k_1 - 1, k_2 - 1)$

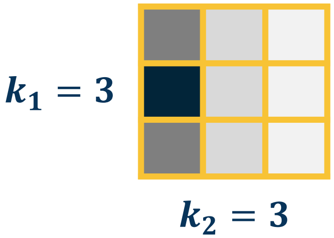


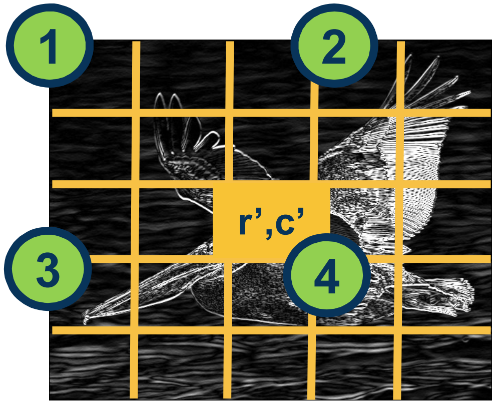


# Extents of Kernel Touching the Pixel

# Extents at the Output







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

This is where the corresponding locations are for the **output**

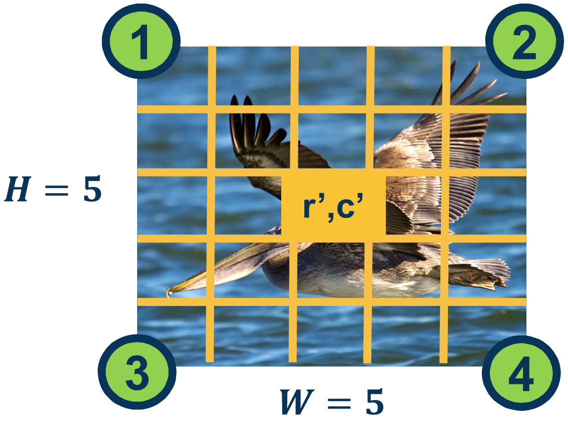
# Summing Gradient Contributions

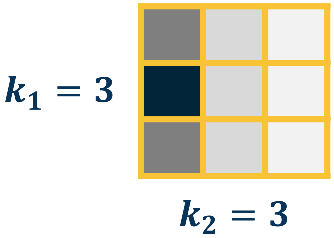
Chain rule for affected pixels (sum gradients):

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

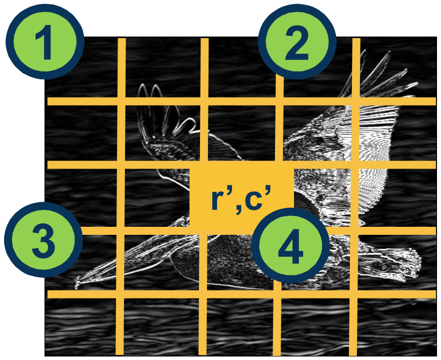
$$\frac{\partial L}{\partial x(r',c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r'-a,c'-b)} \frac{\partial y(r'-a,c'-b)}{\partial x(r',c')}$$

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





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



# Calculating the Gradient

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

$$\boldsymbol{y}(\boldsymbol{r}',\boldsymbol{c}') = (\boldsymbol{x} * \boldsymbol{k})(\boldsymbol{r}',\boldsymbol{c}') = \sum_{\boldsymbol{a}'=0}^{\boldsymbol{k}_1-1} \sum_{\boldsymbol{b}'=0}^{\boldsymbol{k}_2-1} \boldsymbol{x}(\boldsymbol{r}'+\boldsymbol{a}',\boldsymbol{c}'+\boldsymbol{b}')\,\boldsymbol{k}(\boldsymbol{a}',\boldsymbol{b}')$$

Plug in what we actually wanted :

$$\boldsymbol{y}(\boldsymbol{r}'-\boldsymbol{a},\boldsymbol{c}'-\boldsymbol{b}) = (\boldsymbol{x} * \boldsymbol{k})(\boldsymbol{r}',\boldsymbol{c}') = \sum_{\boldsymbol{a}'=0}^{\boldsymbol{k}_1-1} \sum_{\boldsymbol{b}'=0}^{\boldsymbol{k}_2-1} \boldsymbol{x}(\boldsymbol{r}'-\boldsymbol{a}+\boldsymbol{a}',\boldsymbol{c}'-\boldsymbol{b}+\boldsymbol{b}')\,\boldsymbol{k}(\boldsymbol{a}',\boldsymbol{b}')$$

What is $\dfrac{\partial \boldsymbol{y}(\boldsymbol{r}'-\boldsymbol{a},\boldsymbol{c}'-\boldsymbol{b})}{\partial \boldsymbol{x}(\boldsymbol{r}',\boldsymbol{c}')} = \mathbf{k}(\boldsymbol{a},\boldsymbol{b})$

(we want term with $\boldsymbol{x}(\boldsymbol{r}',\boldsymbol{c}')$ in it;
this happens when $\mathbf{a}' = \mathbf{a}$ and $\mathbf{b}' = \mathbf{b}$

Plugging in to earlier equation:

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

**Does this look familiar?**

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

**Convolution between upstream gradient and kernel!**

**(can implement by flipping kernel and cross- correlation)**

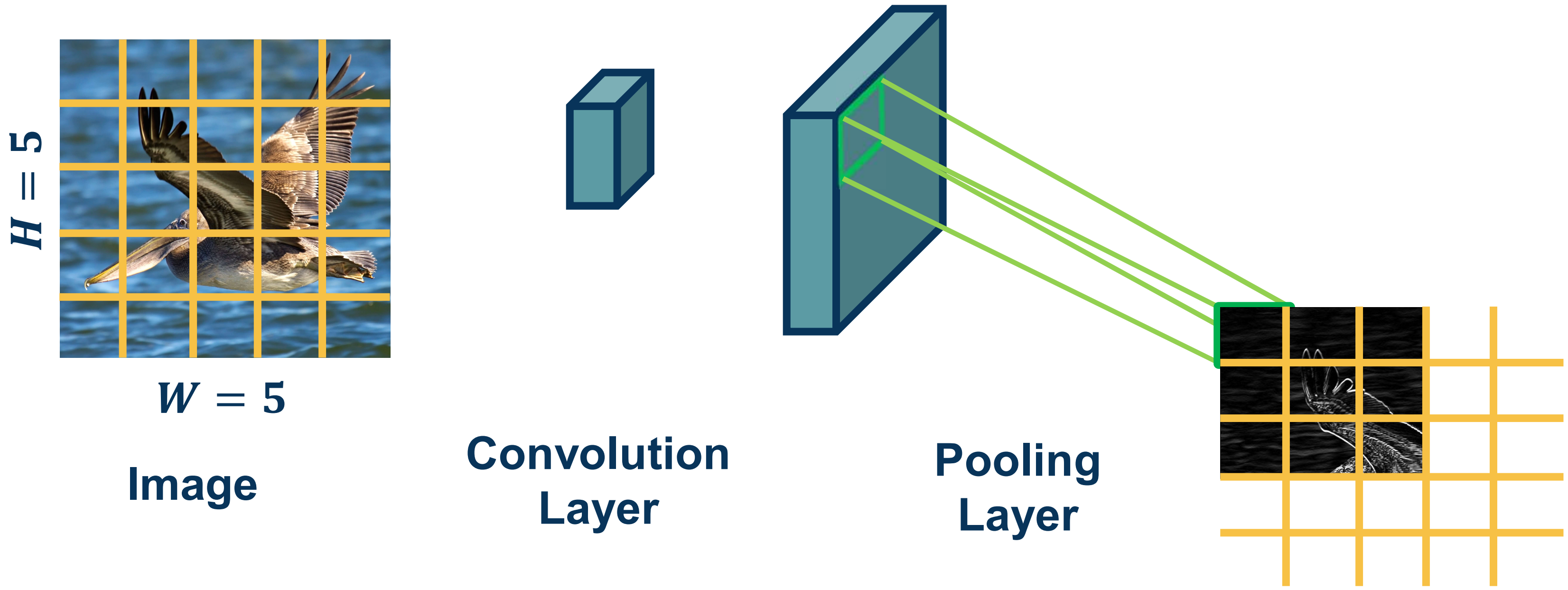
**Again, all operations can be implemented via matrix multiplications (same as FC layer)!**

# Backwards is Convolution

# Simple Convolutional Neural Networks

# Combining Convolution & Pooling Layers

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



$H = 5$

$W = 5$

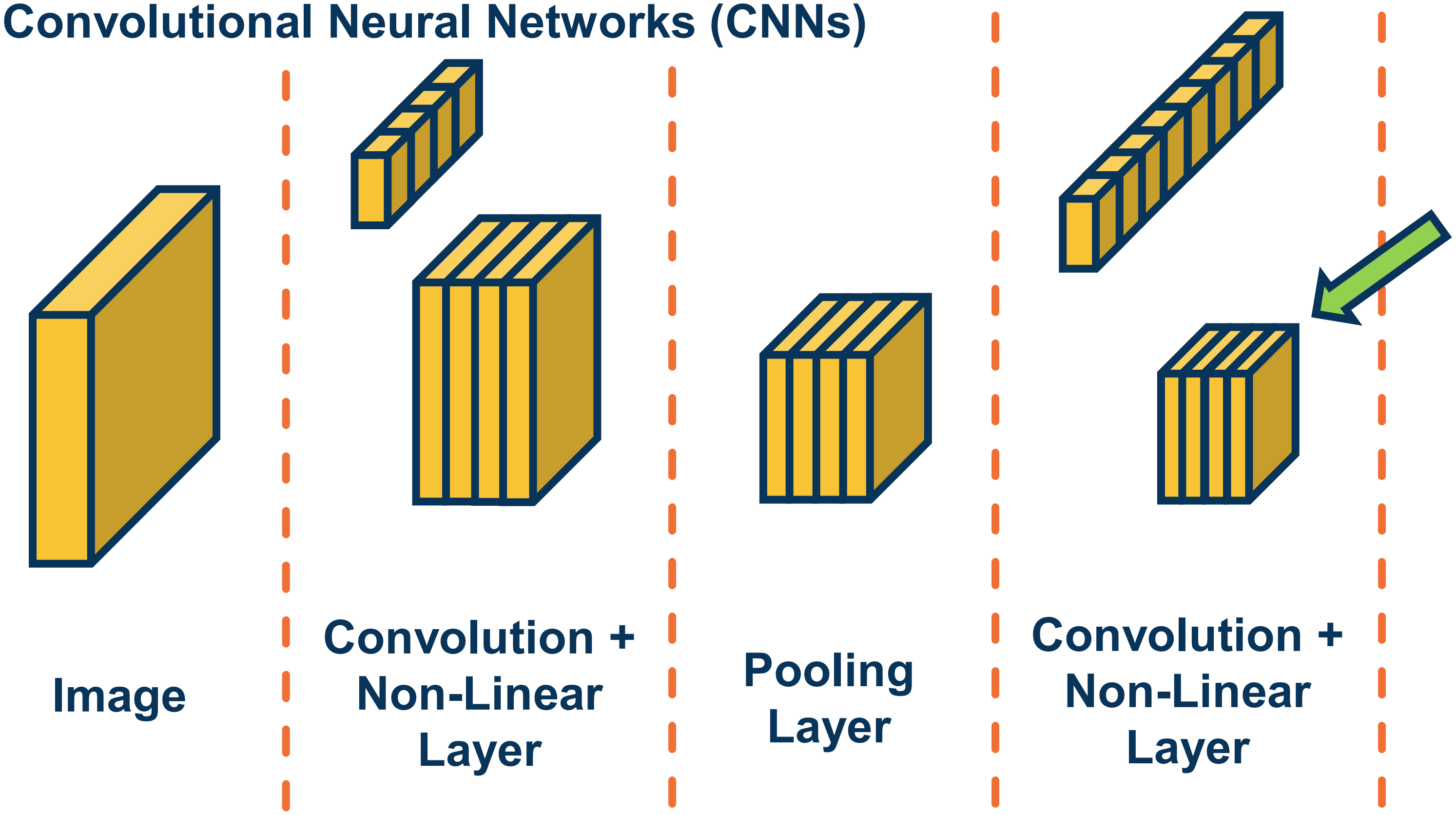
**Image**

**Convolution Layer**

**Pooling Layer**

## Alternating Convolution and Pooling

**Convolutional Neural Networks (CNNs)**



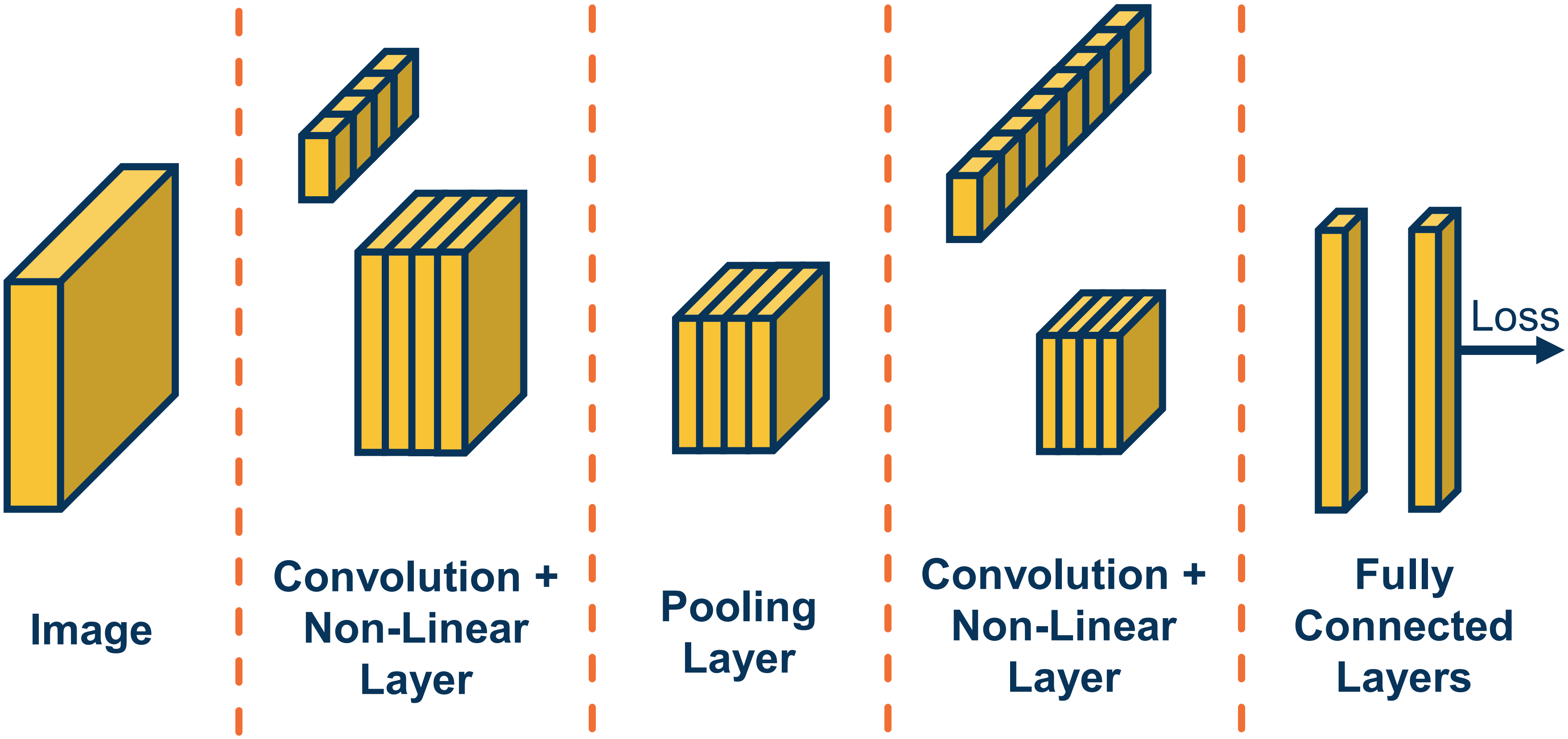
Image

Convolution + Non-Linear Layer

Pooling Layer

Convolution + Non-Linear Layer

Useful, lower-dimensional features



**Image**

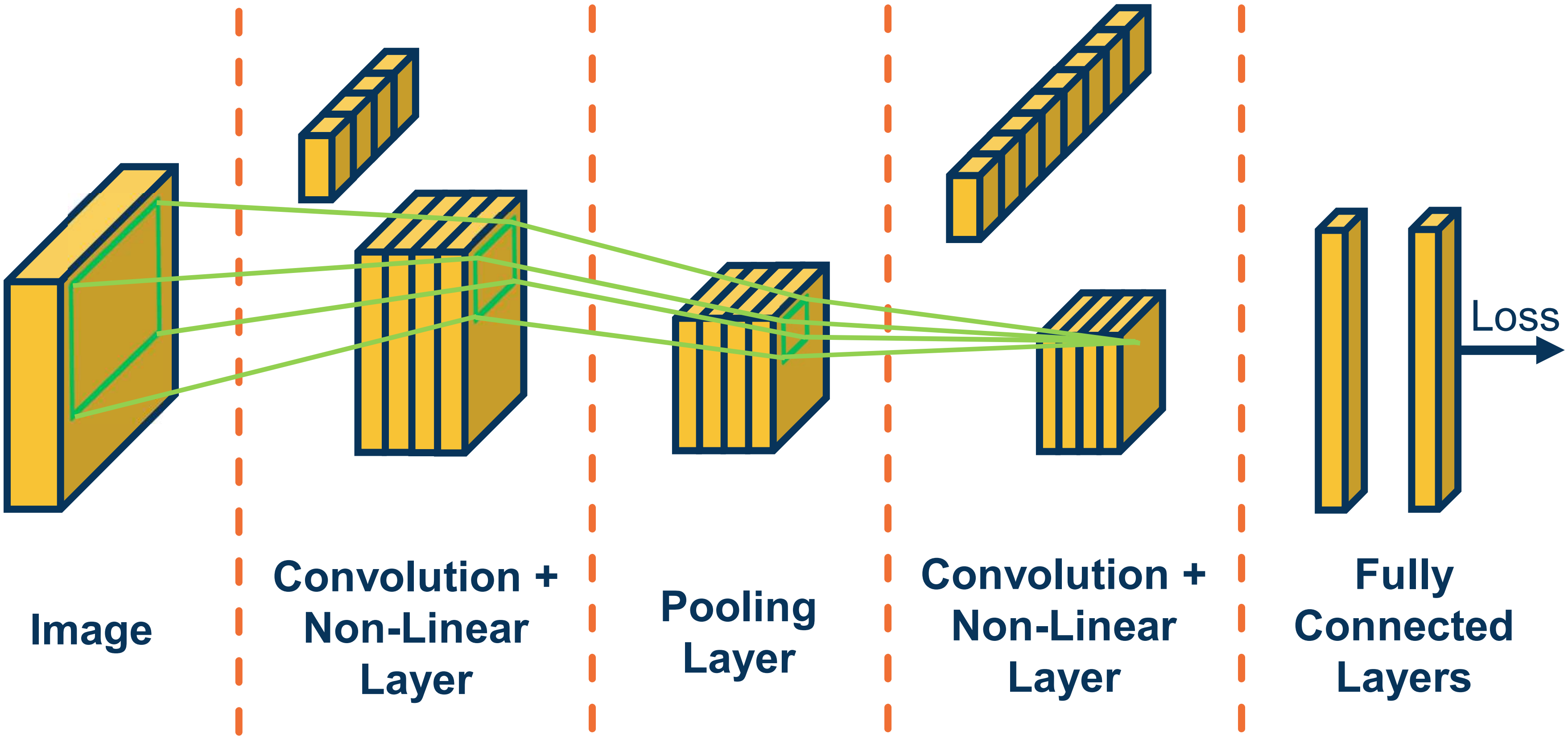
**Convolution + Non-Linear Layer**

**Pooling Layer**

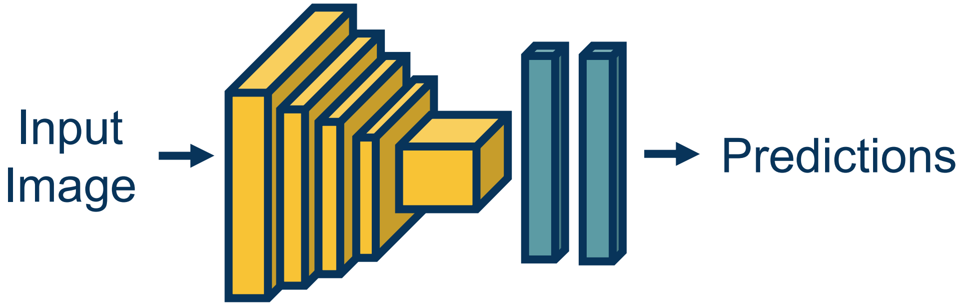
**Convolution + Non-Linear Layer**

**Fully Connected Layers**

# Adding a Fully Connected Layer

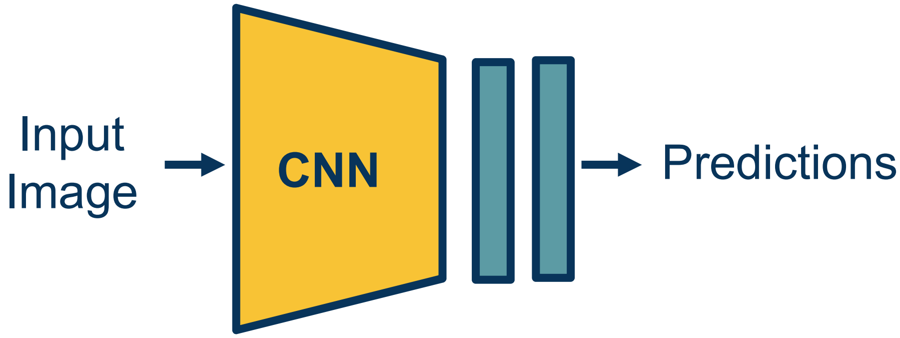


# Receptive Fields


Input Image

Predictions

**Convolutional Neural Networks**


Input Image

CNN

Predictions

# Typical Depiction of CNNs

# These architectures have existed **since 1980s**

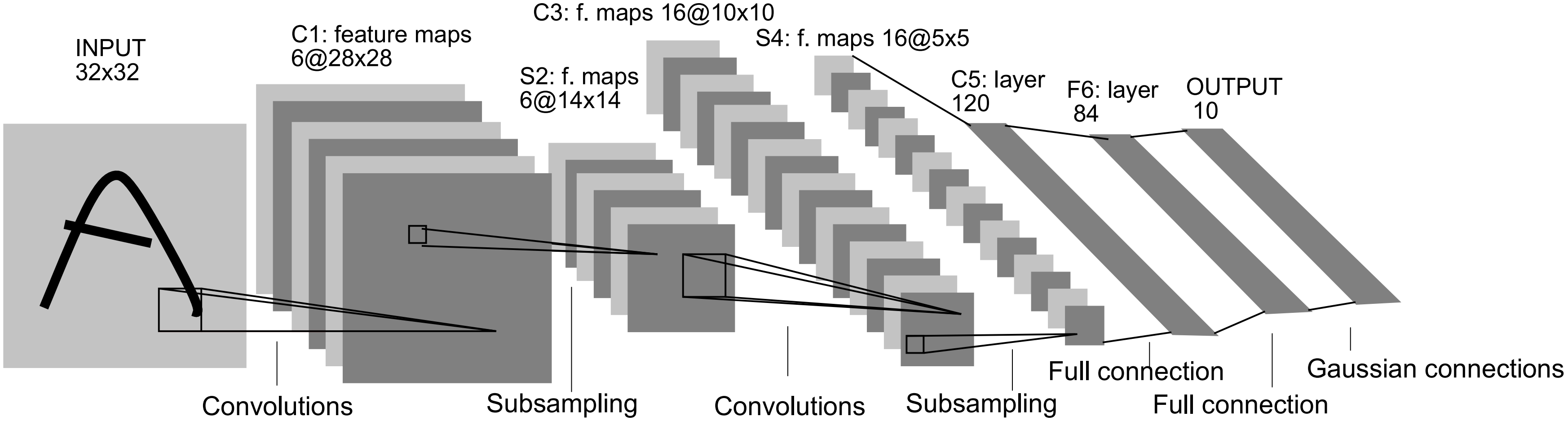


*Image Credit: Yann LeCun, Kevin Murphy*

## LeNet Architecture

# Handwriting Recognition

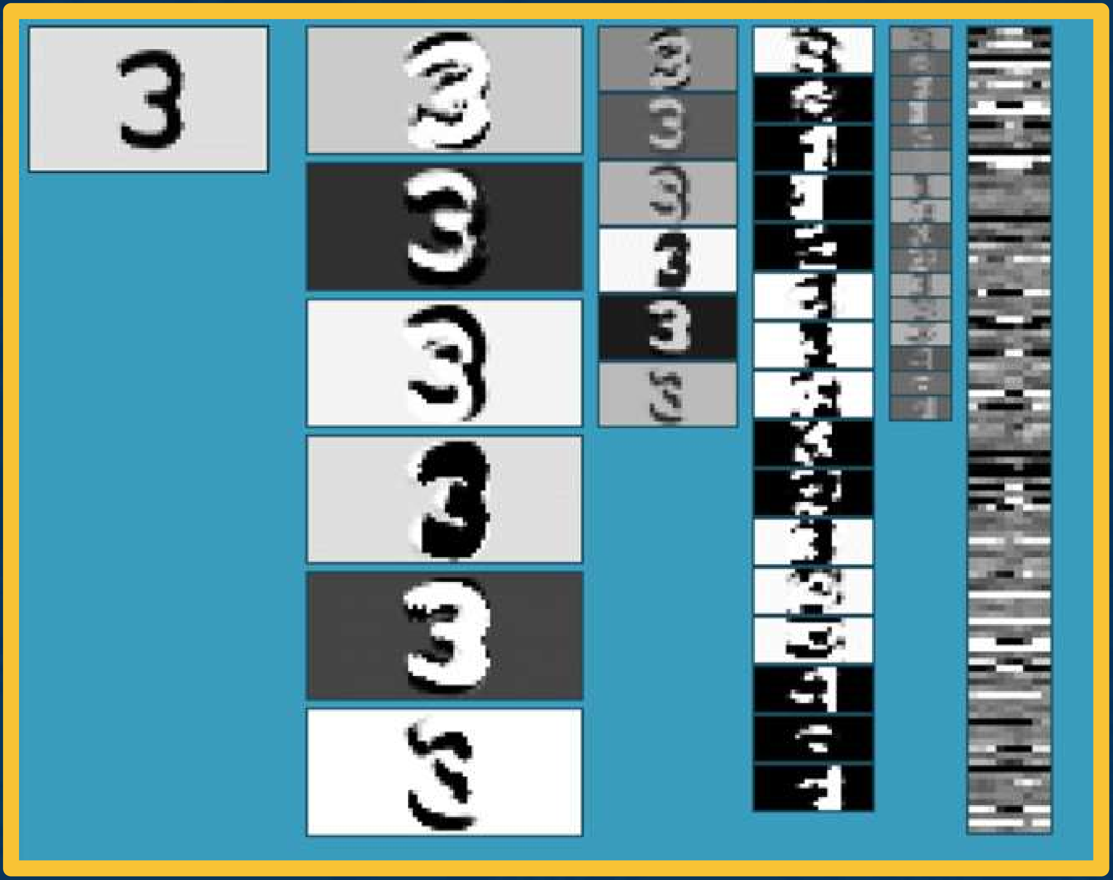


*Image Credit: Yann LeCun*

# Translation Equivariance (Conv Layers) & Invariance (Output)

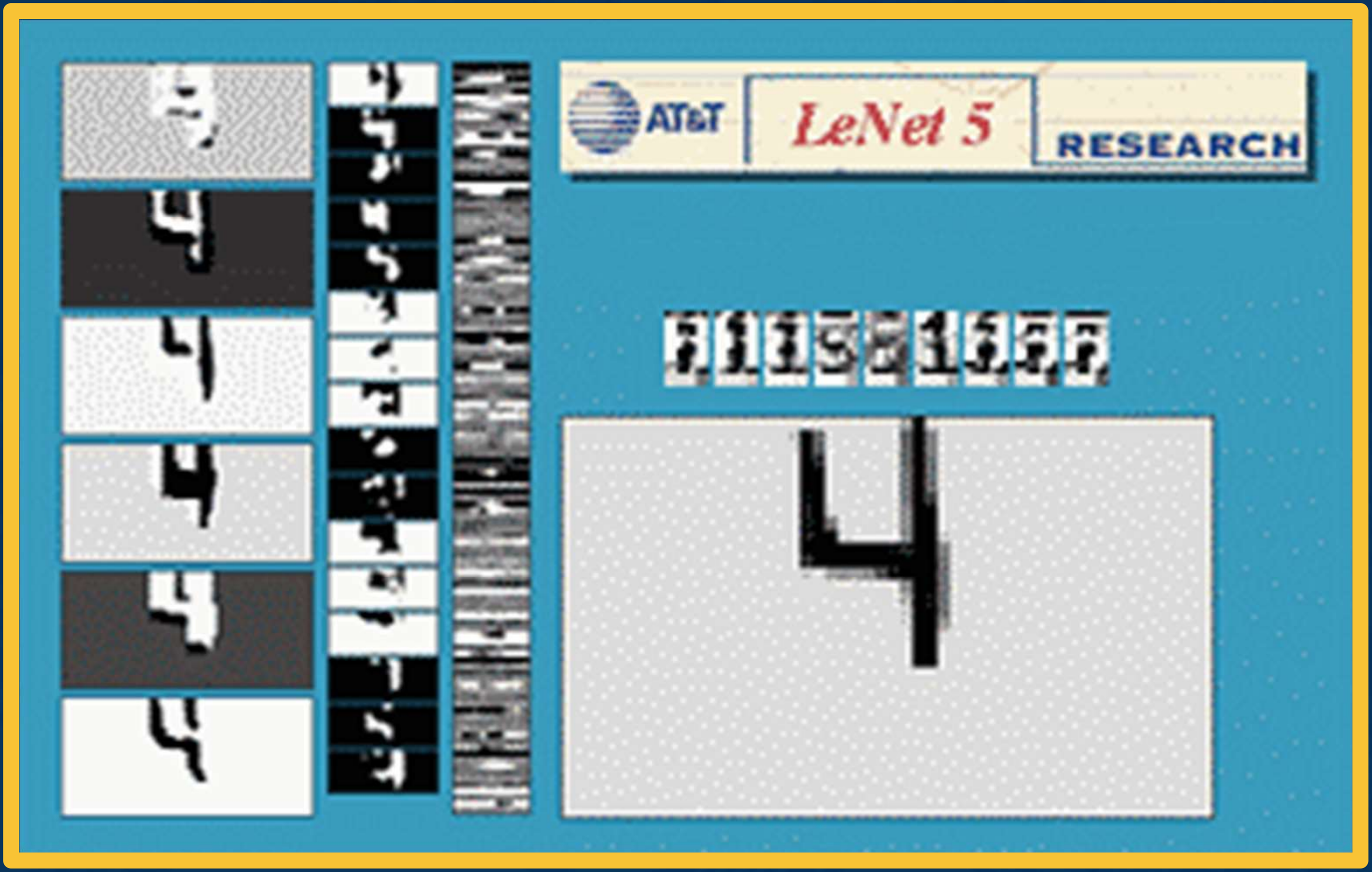


Image Credit:
Yann LeCun

# (Some) Rotation Invariance

Image Credit:
Yann LeCun

# (Some) Scale Invariance

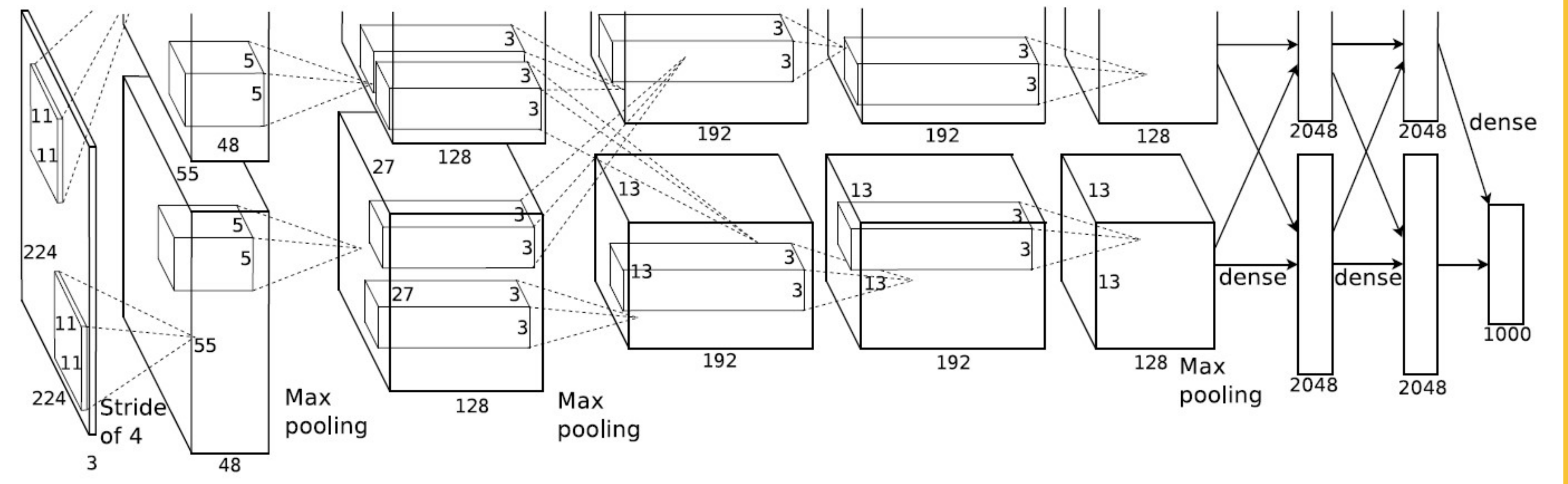
Image Credit:
Yann LeCun

# Advanced Convolutional Networks

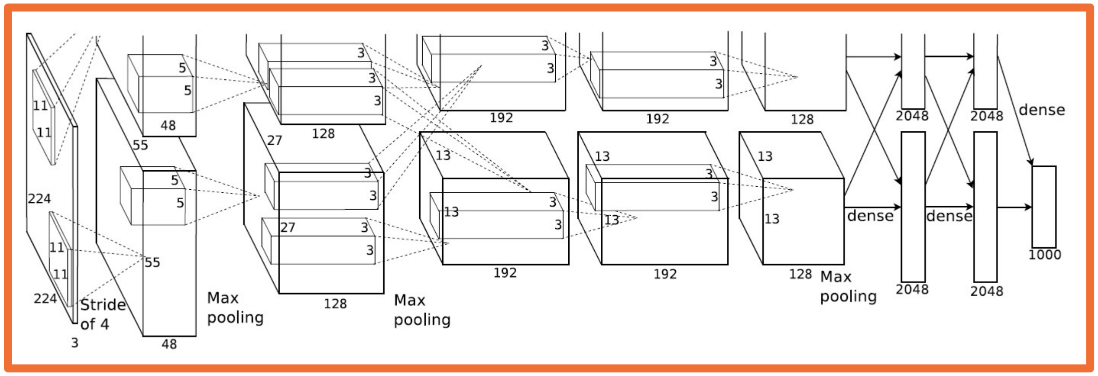
# The Importance of Benchmarks

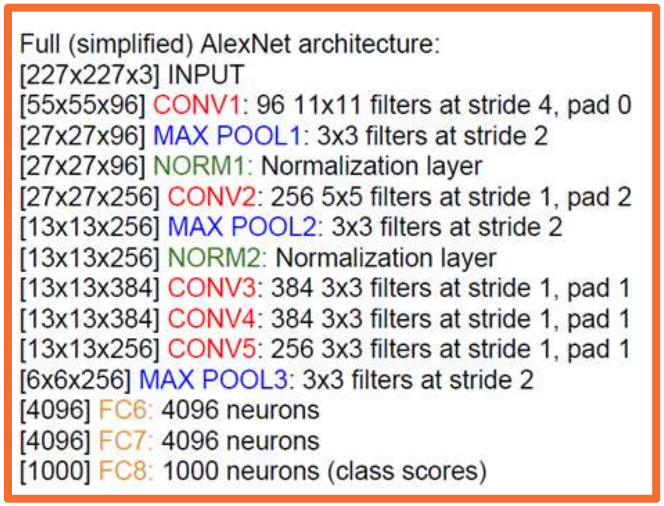
*From: https://paperswithcode.com*

# AlexNet - Architecture



*From: Krizhevsky et al., ImageNet Classification with Deep ConvolutionalNeural 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
[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)

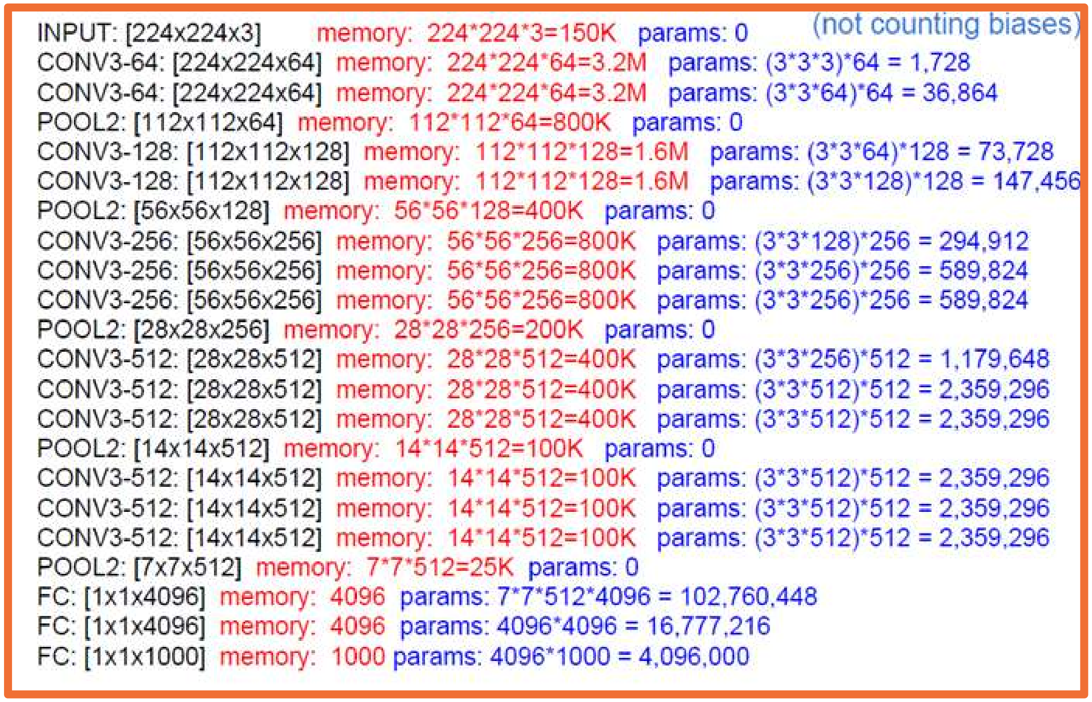
**Key aspects:**

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

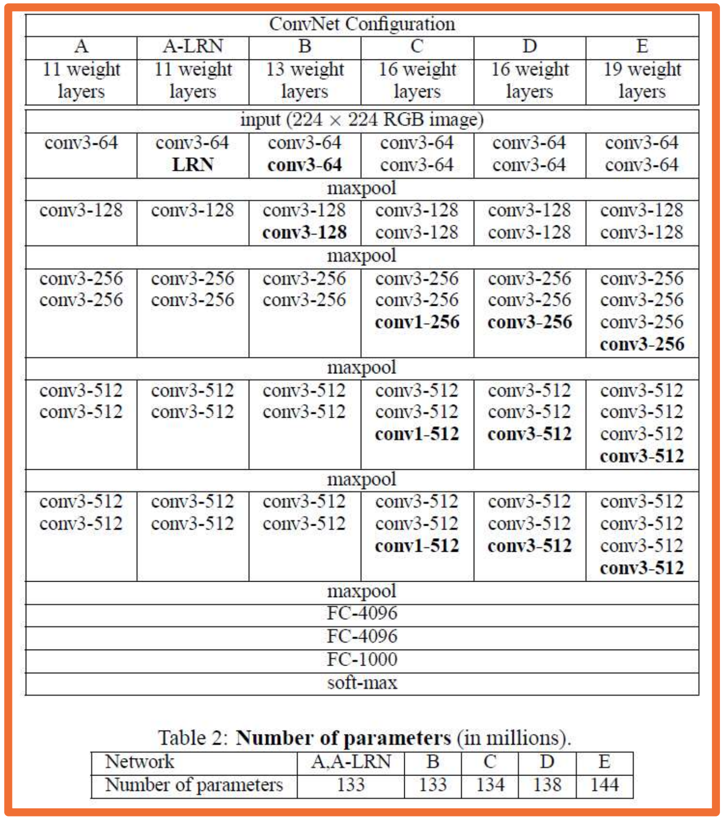
*From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

# AlexNet – Layers and Key Aspects

# VGG



```
INPUT: [224x224x3]        memory:  224*224*3=150K   params: 0          (not counting biases)
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   params: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   params: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   params: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   params: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   params: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   params: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   params: (3*3*512)*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)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  params: 0
FC: [1x1x4096]  memory:  4096  params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  params: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 params: 4096*1000 = 4,096,000
```



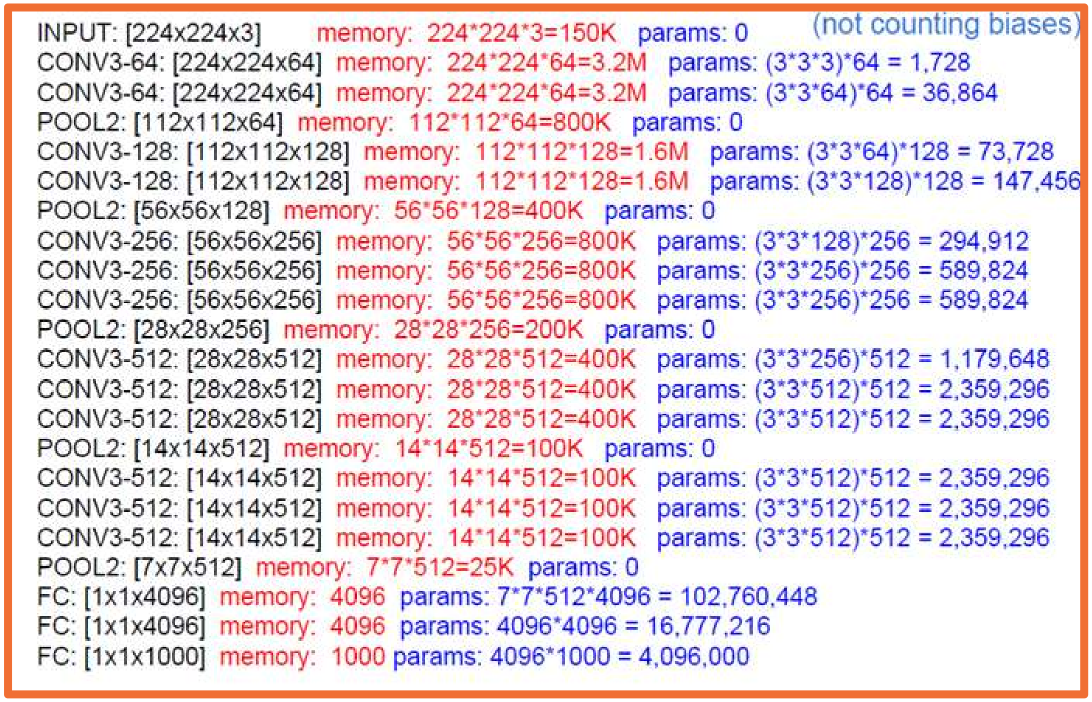
| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224 × 224 RGB image) | | | | | |
| conv3-64 | conv3-64<br>**LRN** | conv3-64<br>**conv3-64** | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128<br>**conv3-128** | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 |
| maxpool | | | | | |
| conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>**conv1-256** | conv3-256<br>conv3-256<br>**conv3-256** | conv3-256<br>conv3-256<br>conv3-256<br>**conv3-256** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

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

| Network | A,A-LRN | B | C | D | E |
|---|---|---|---|---|---|
| Number of parameters | 133 | 133 | 134 | 138 | 144 |

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

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



(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*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)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

**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*

# Parameters and Memory

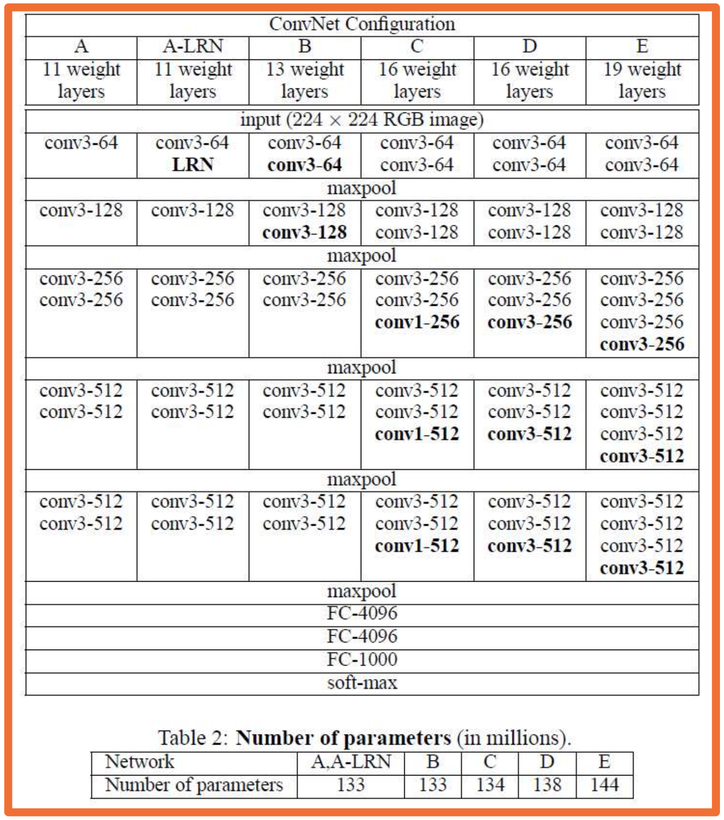
# VGG – Key Characteristics

## Key aspects:

Repeated application of:

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

Very large number of parameters



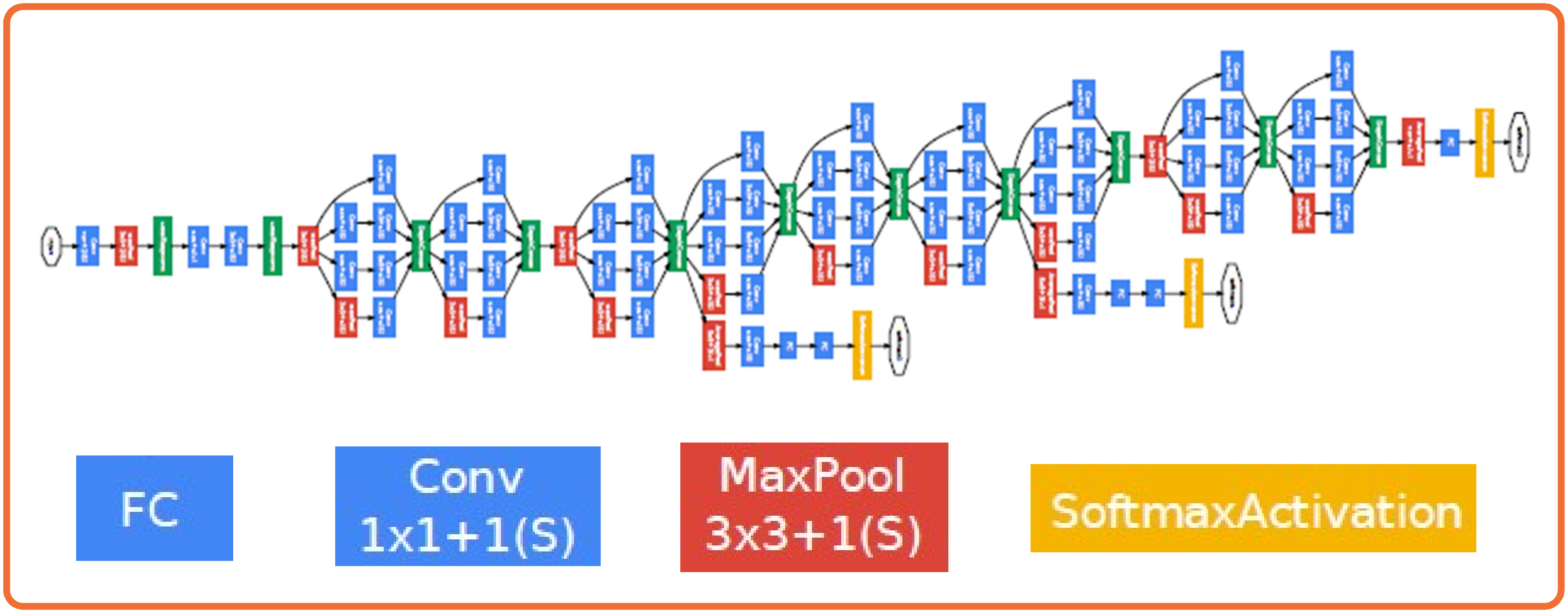
| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224 × 224 RGB image) | | | | | |
| conv3-64 | conv3-64<br>**LRN** | conv3-64<br>**conv3-64** | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128<br>**conv3-128** | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 |
| maxpool | | | | | |
| conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>**conv1-256** | conv3-256<br>conv3-256<br>**conv3-256** | conv3-256<br>conv3-256<br>conv3-256<br>**conv3-256** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

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

| Network | A,A-LRN | B | C | D | E |
|---|---|---|---|---|---|
| Number of parameters | 133 | 133 | 134 | 138 | 144 |

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

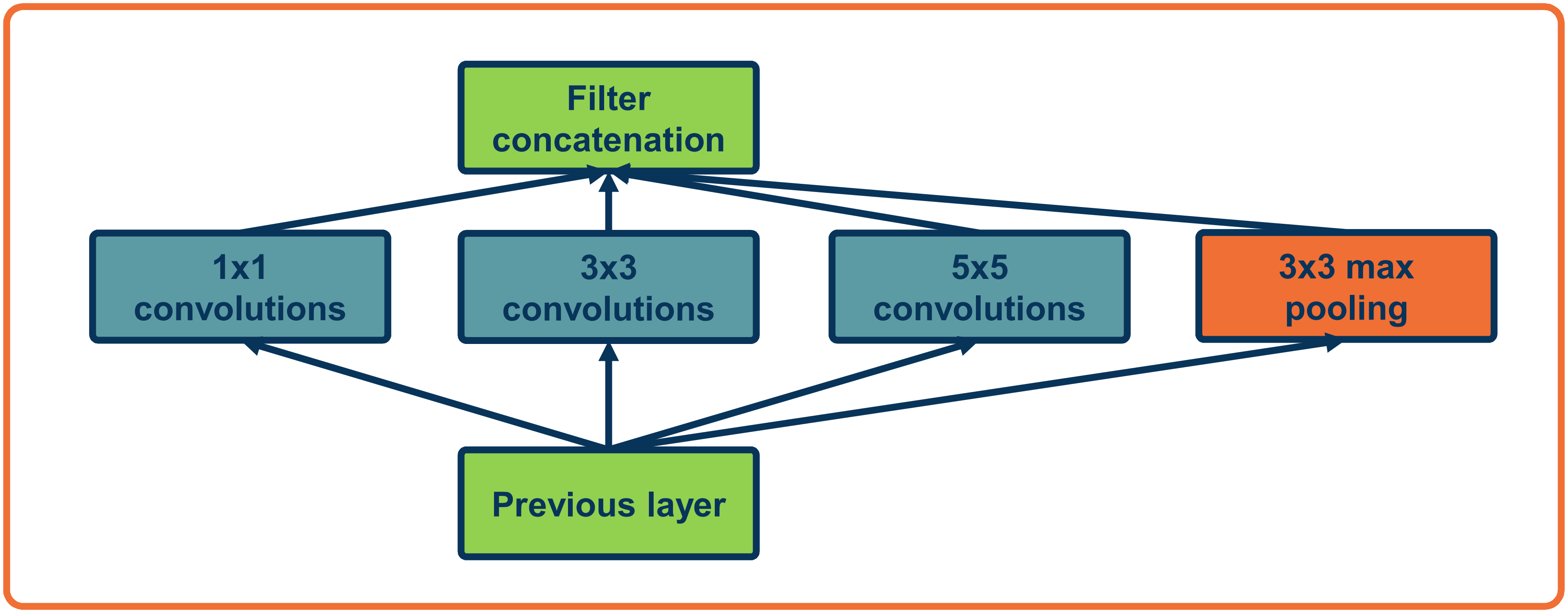
## But have become **deeper and more complex**



*From: Szegedy et al. Going deeper with convolutions*

# Inception Architecture

**Key idea:** Repeated blocks and multi-scale features


Filter concatenation

1x1 convolutions

3x3 convolutions

5x5 convolutions

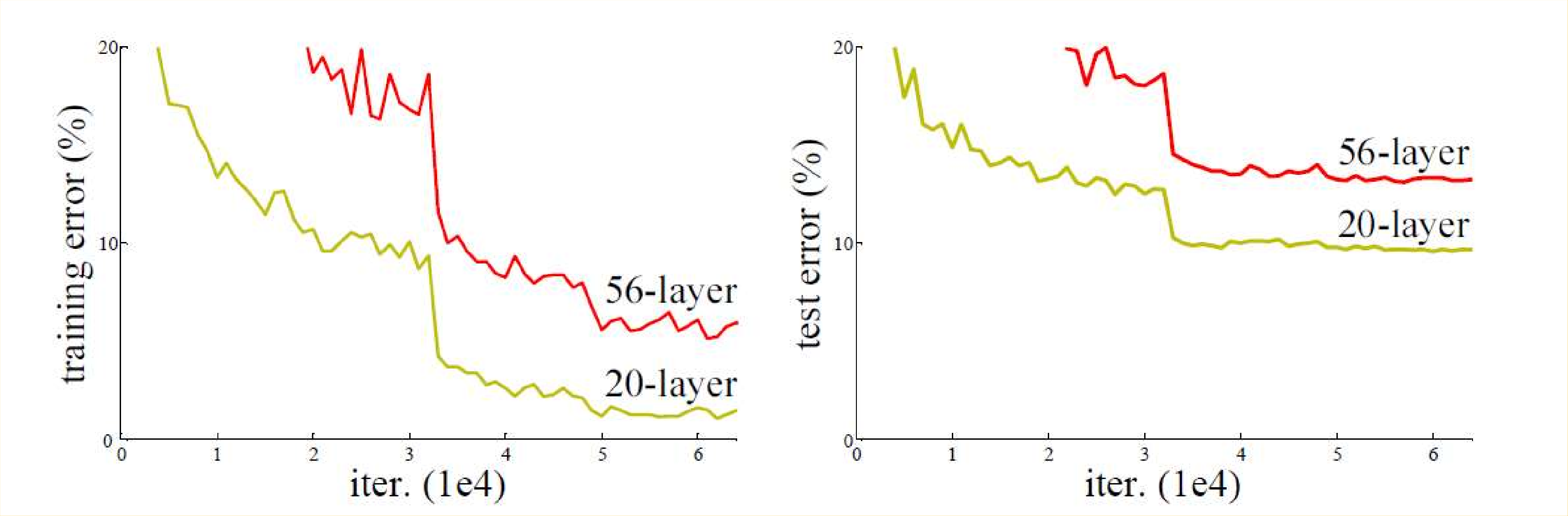
3x3 max pooling

Previous layer

*From: Szegedy et al. Going deeper with convolutions*

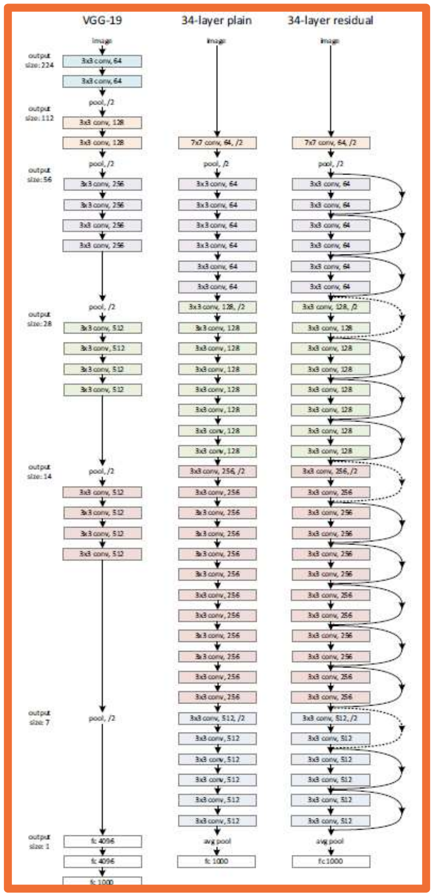
# Inception Module

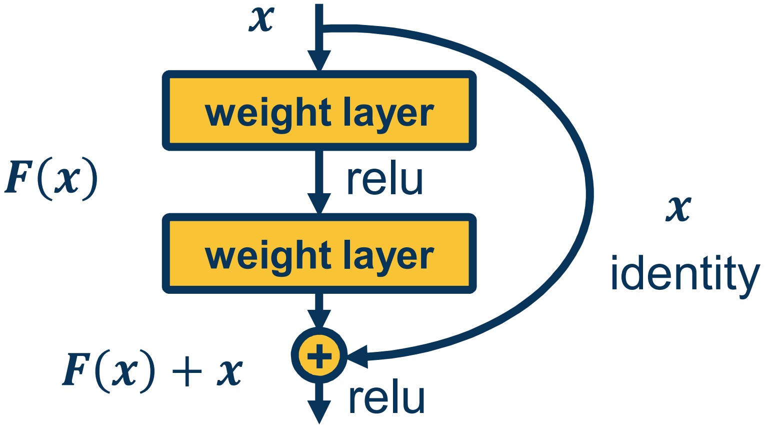
# The Challenge of Depth



*From: He et al., Deep Residual Learning for Image Recognition*

Optimizing very deep networks is challenging!





$x$

**weight layer**

$F(x)$

relu

**weight layer**

$x$

identity

$F(x) + x$

relu

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

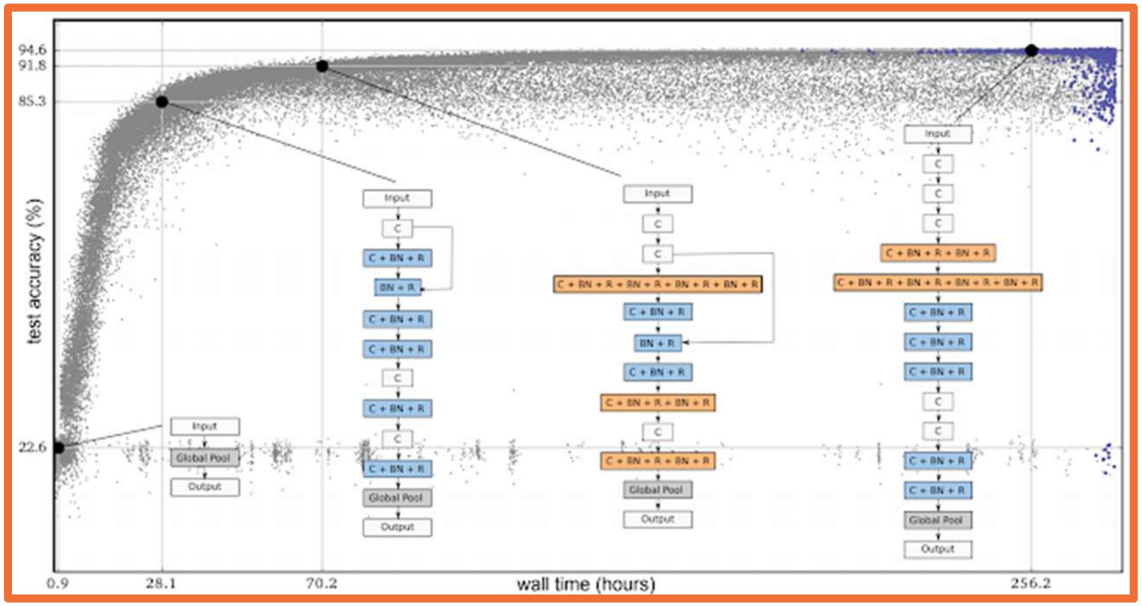
Same is true for gradients!

*From: He et al., Deep Residual Learning for Image Recognition*

# Residual Blocks and Skip Connections

**Several ways to *learn* architectures:**

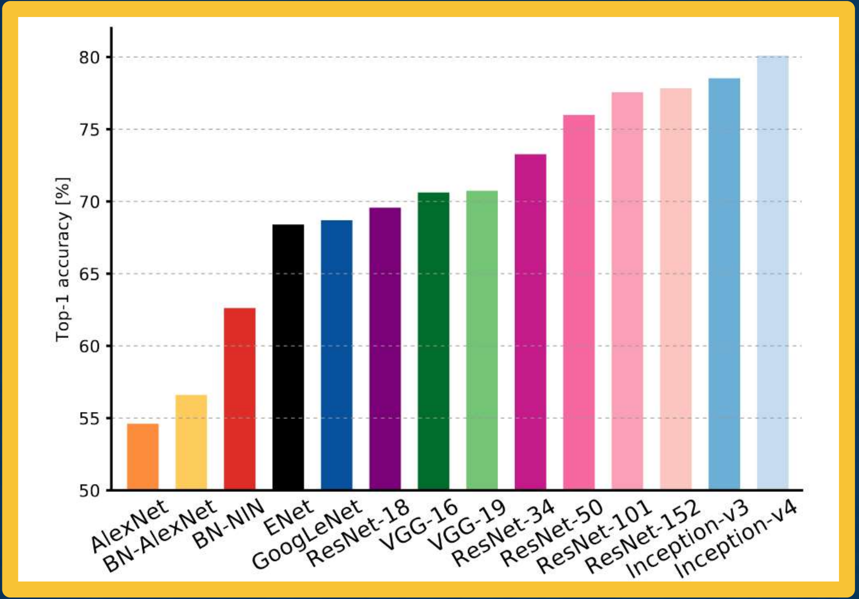
- Evolutionary learning and reinforcement learning
- Prune over-parameterized networks
- Learning of **repeated blocks** typical

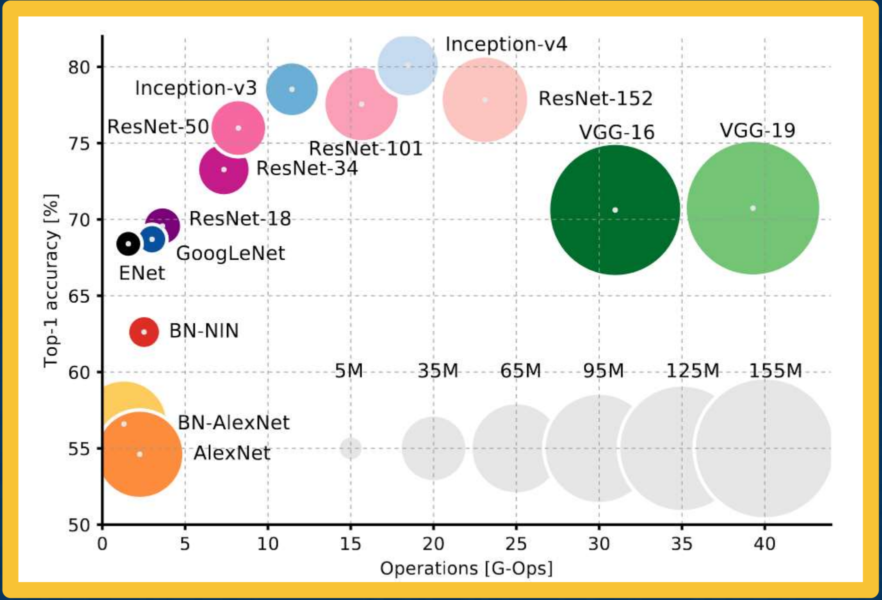


*From: https://ai.googleblog.com/2018/03/using-evolutionary-automl-to-discover.html*

## Evolving Architectures and AutoML

# Computational Complexity

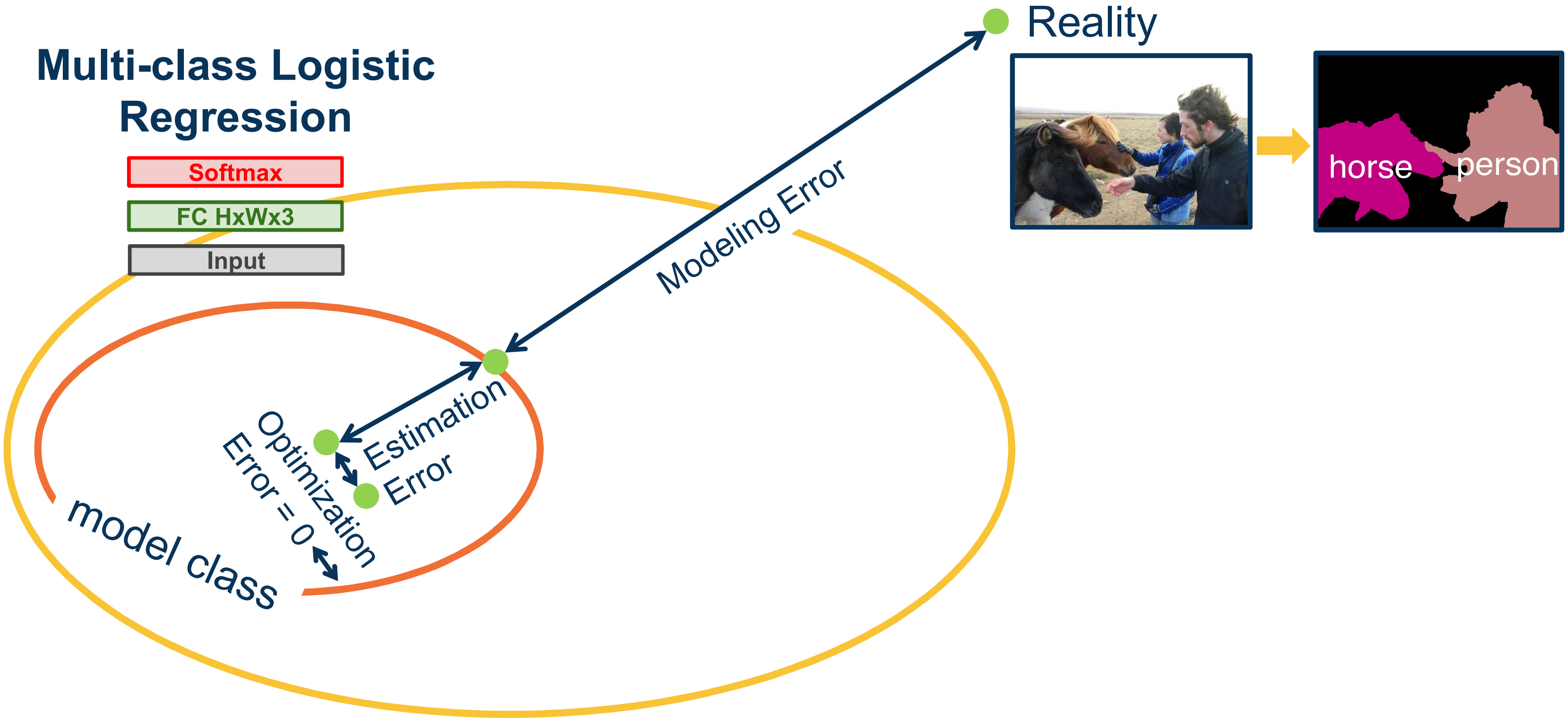




*From: An Analysis Of Deep Neural Network Models For Practical Applications*

# Transfer Learning & Generalization

# Generalization



**Multi-class Logistic Regression**

Softmax

FC HxWx3

Input

Reality

horse

person

Modeling Error

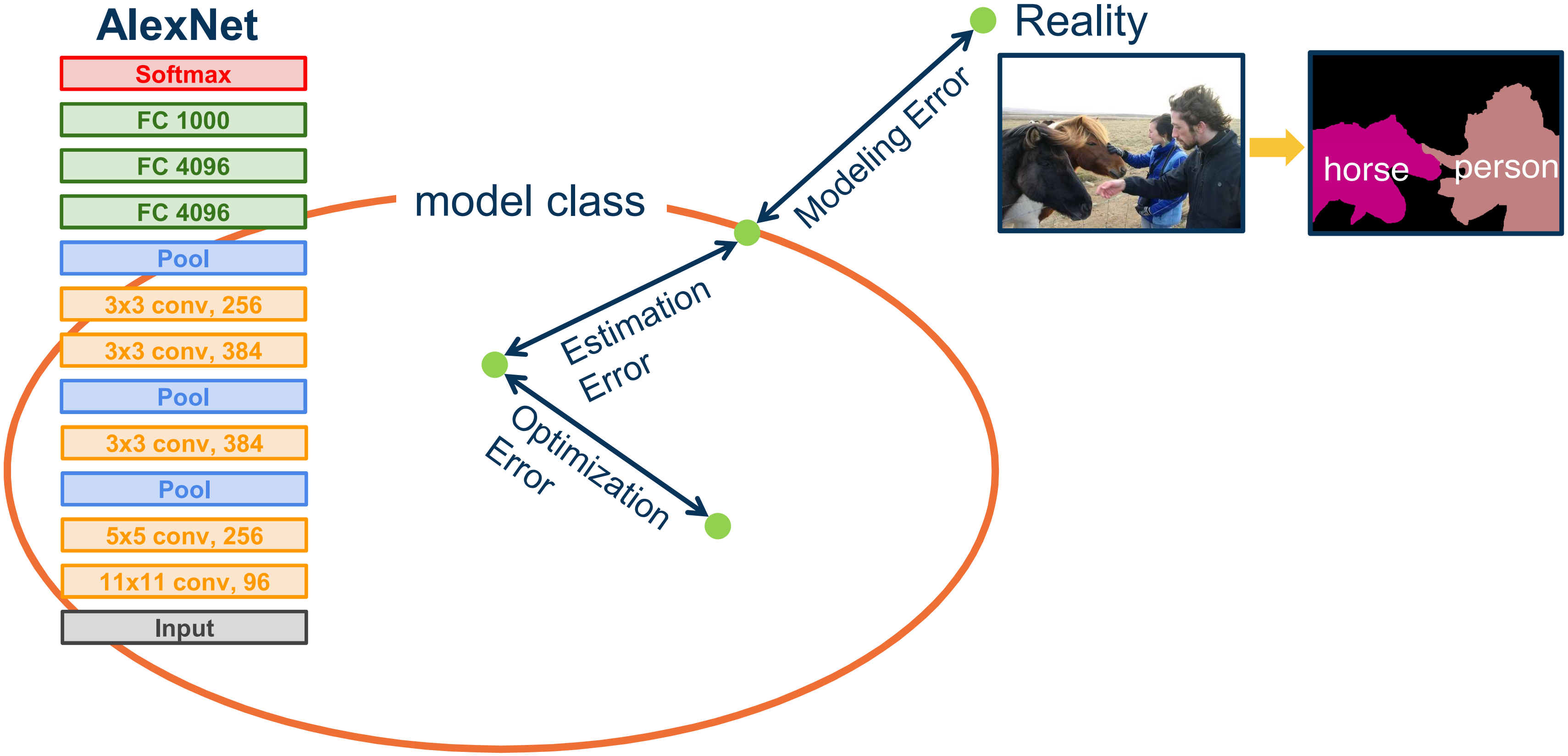
Estimation Error

Optimization Error = 0

model class

*From: slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

# Generalization



**AlexNet**

- Softmax
- FC 1000
- FC 4096
- FC 4096
- Pool
- 3x3 conv, 256
- 3x3 conv, 384
- Pool
- 3x3 conv, 384
- Pool
- 5x5 conv, 256
- 11x11 conv, 96
- Input

model class

Reality

Modeling Error

Estimation Error

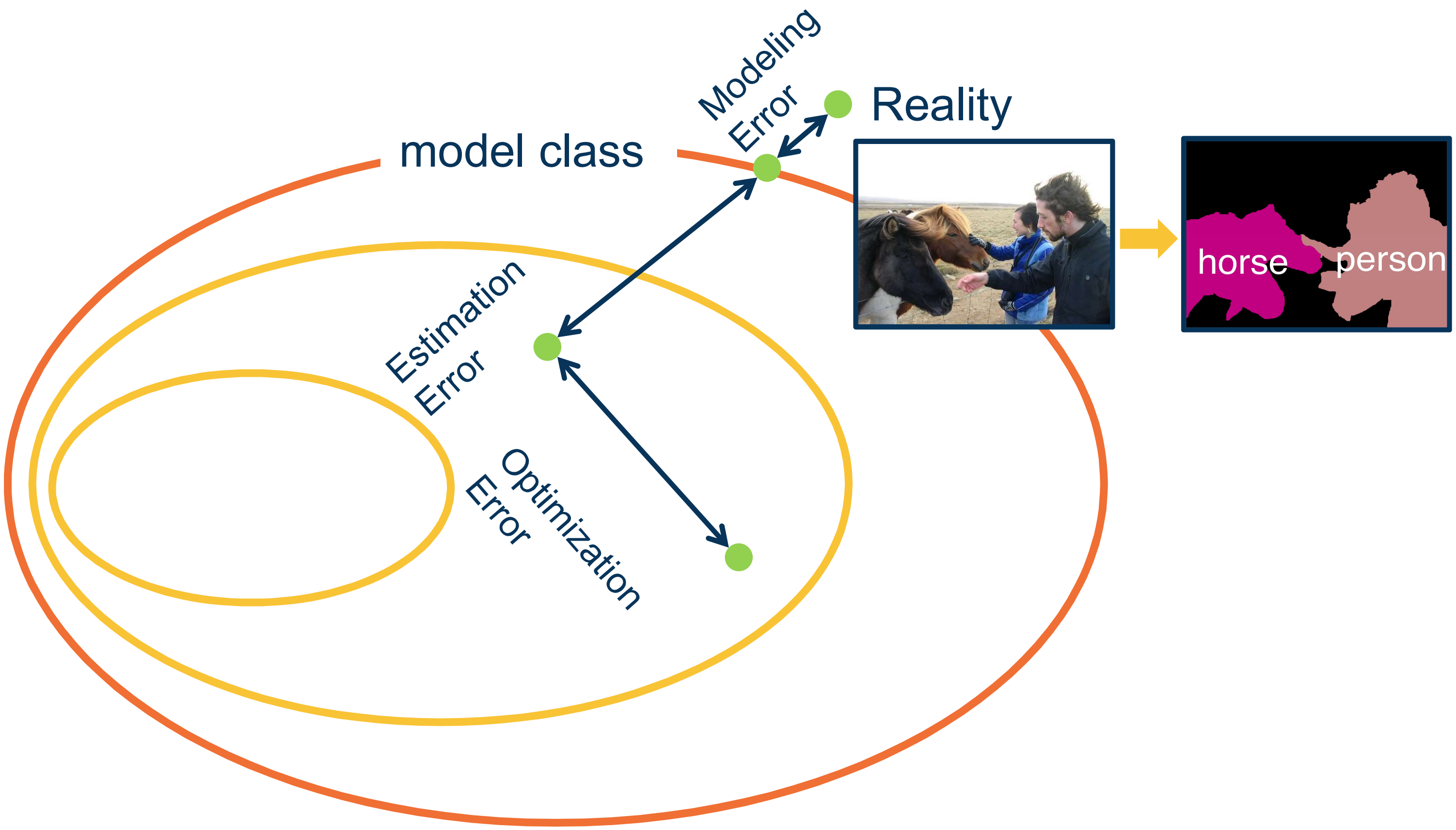
Optimization Error

*From: slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

# Generalization

**VGG19**

Softmax
FC 1000
FC 4096
FC 4096
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 256
3x3 conv, 256
Pool
3x3 conv, 128
3x3 conv, 128
Pool
3x3 conv, 64
3x3 conv, 64
Input

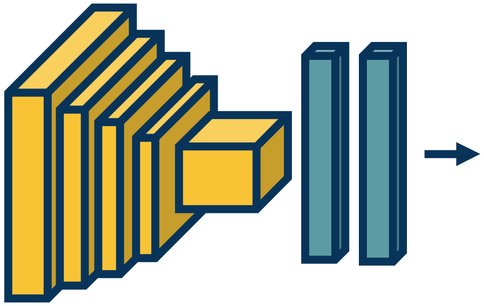


*From: slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

# Transfer Learning – Training on Large Dataset

**What if we don’t have enough data?**

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

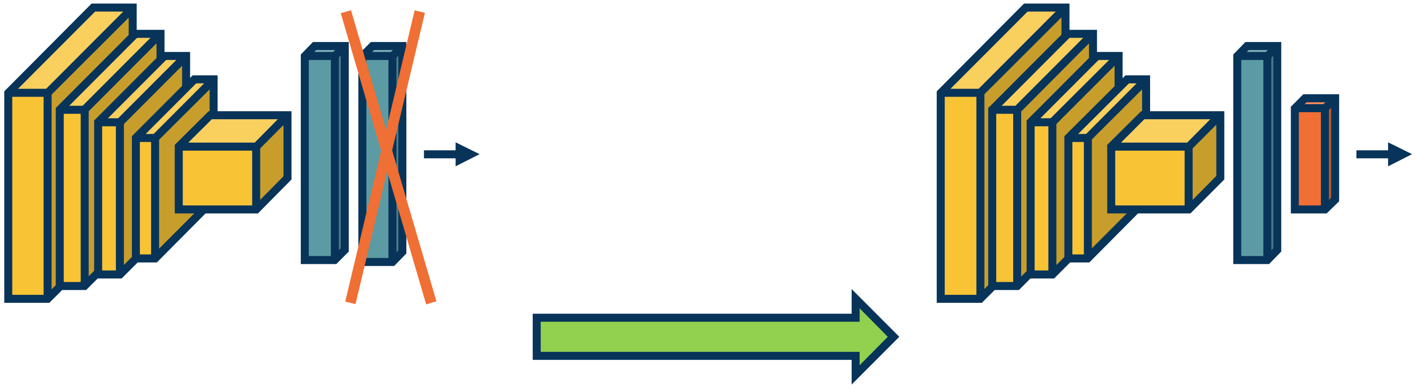


Input Image

Predictions

**Convolutional Neural Networks**

**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

## Initializing with Pre-Trained Network

**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)

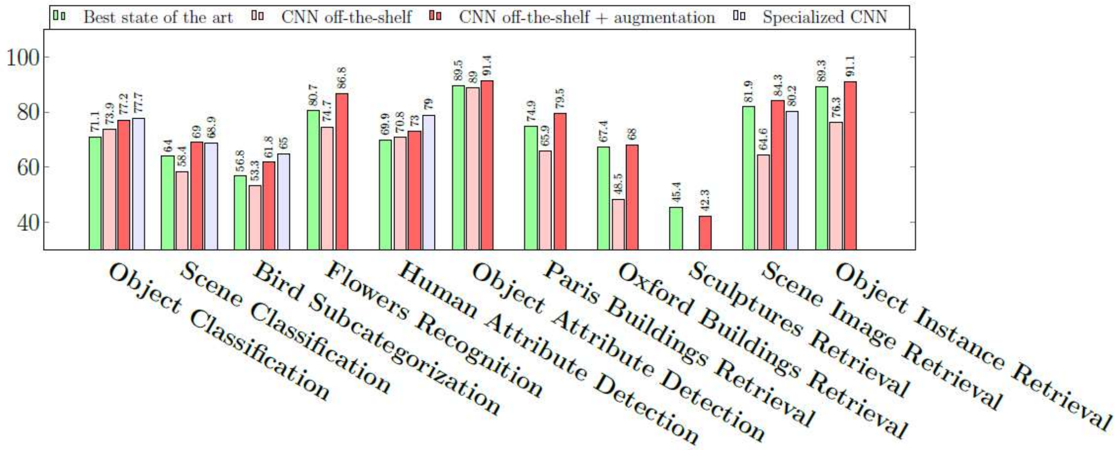
Replace last layer with new fully-connected for output nodes per new category

# Finetuning on New Dataset

# Surprising Effectiveness of Transfer Learning

**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)



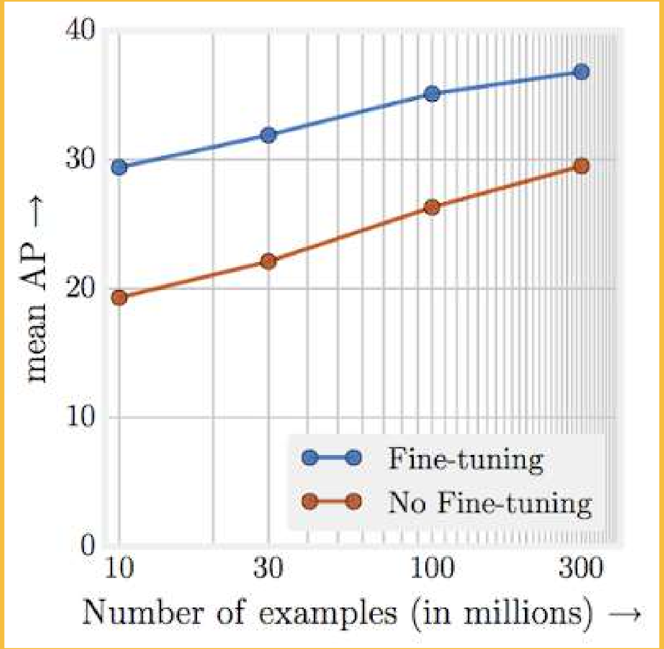
*From: Razavian et al., CNN Features off-the-shelf: an Astounding Baseline for Recognition*

# Learning with Less Labels

**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 https://ai.googleblog.com/2017/07/revisiting-unreasonable-effectiveness.html*

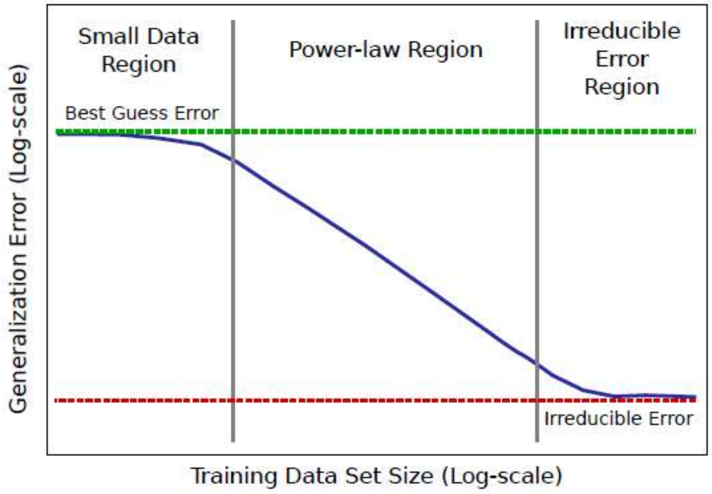


Figure 6: Sketch of power-law learning curves

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

## There is a large number of different low-labeled settings in DL research

| Setting | Source | Target | Shift Type |
|---|---|---|---|
| Semi-supervised | Single labeled | Single unlabeled | None |
| Domain Adaptation | Single labeled | Single unlabeled | Non-semantic |
| Domain Generalization | Multiple labeled | Unknown | Non-semantic |
| Cross-Category Transfer | Single labeled | Single unlabeled | Semantic |
| Few-Shot Learning | Single labeled | Single few-labeled | Semantic |
| Un/Self-Supervised | Single unlabeled | Many labeled | Both/Task |

**Non-Semantic Shift**

**Semantic Shift**

# Dealing with Low-Labeled Situations