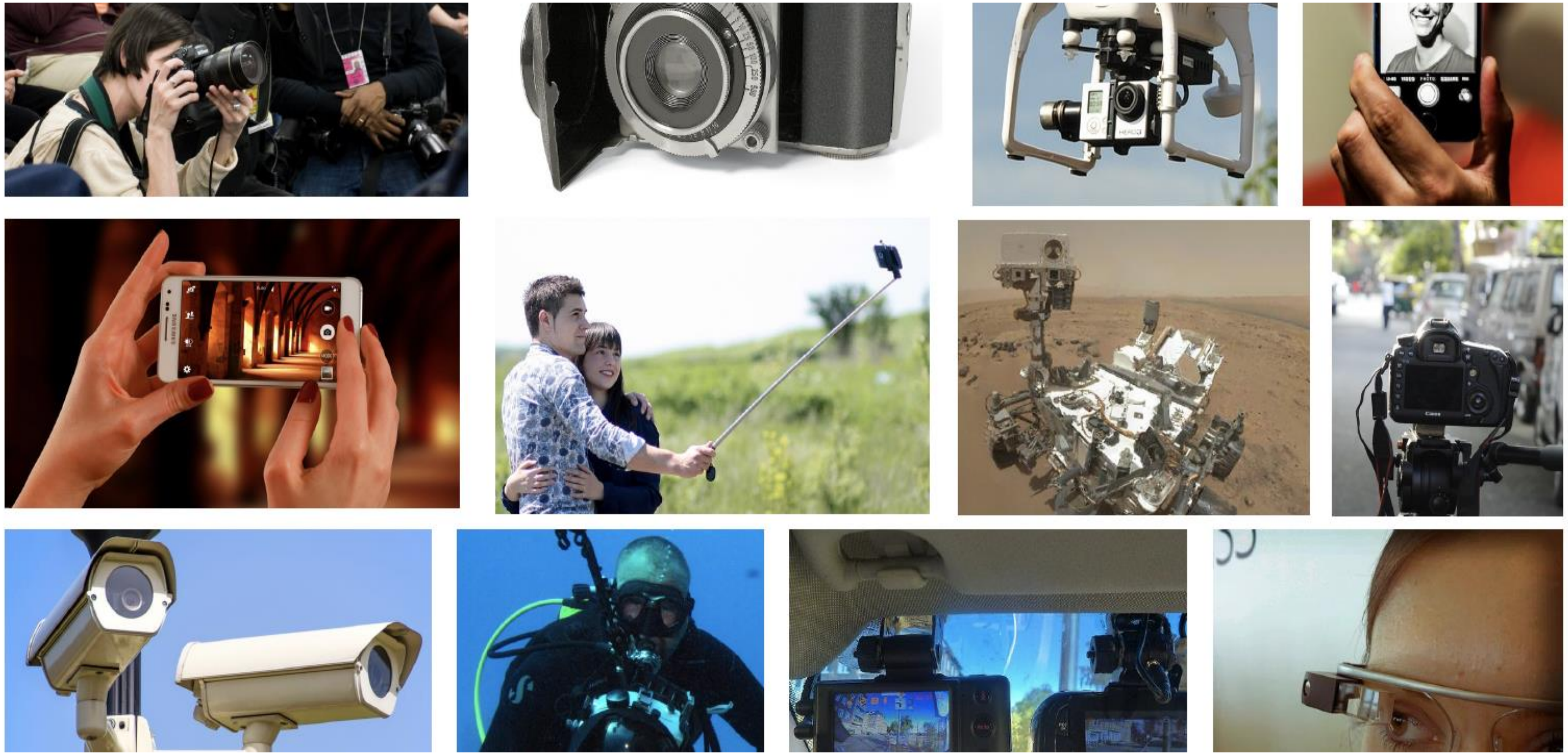
Bios 740- Chapter 3. Convolutional Neural Networks (CNN)

Acknowledgement: Thanks to Miss Xiaoqi Li for preparing some of the slides! I copied some pictures from the Wikipedia and the lecture pdf files of StanfordCS231n and Prof. Manolis Kellis.

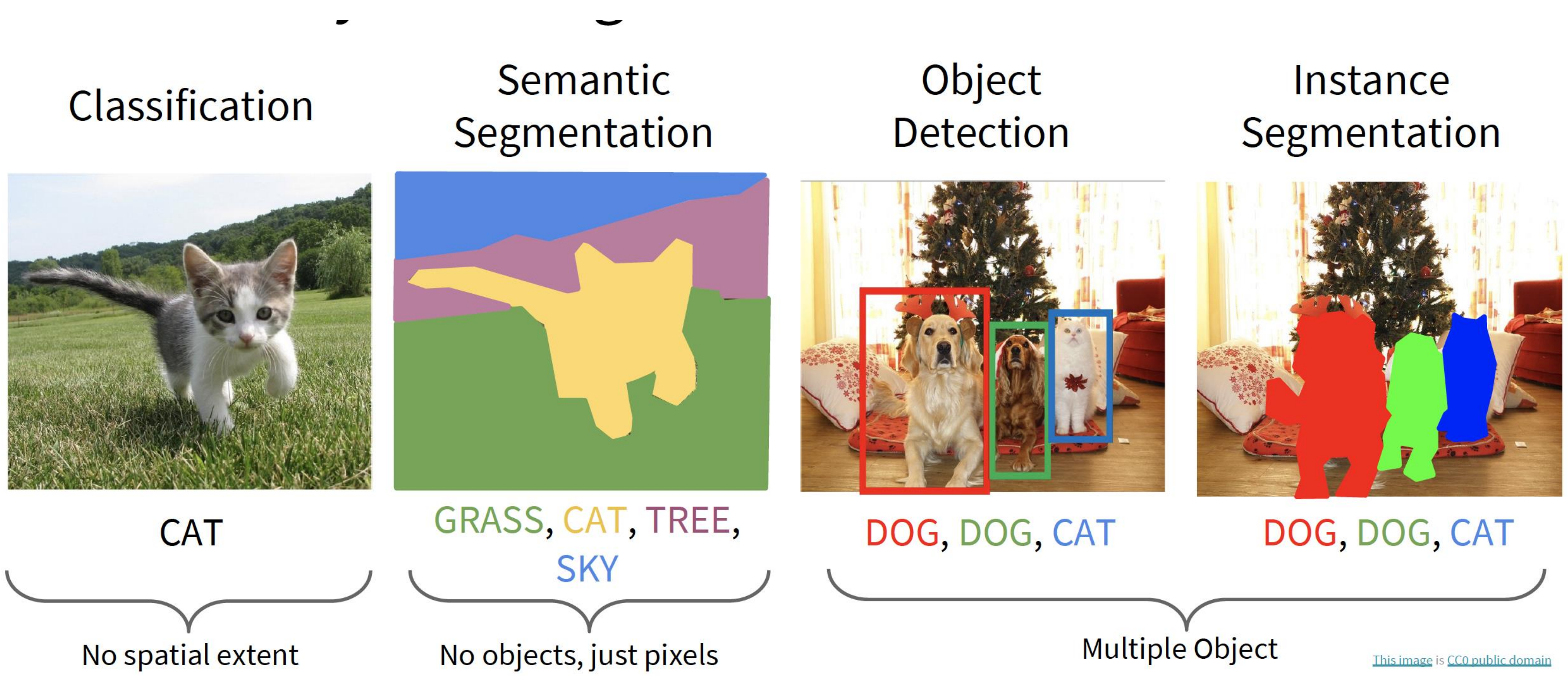
Content

Content

Nature Image Data is Everywhere



Major CV Tasks

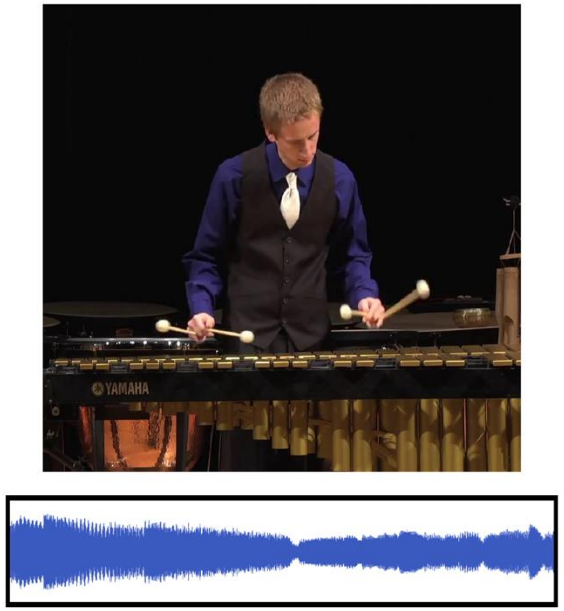


Other CV Tasks

Video Classification

Running? Jumping?

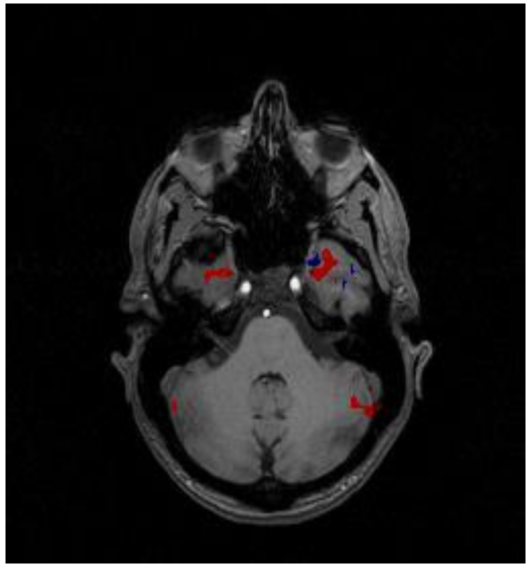
Multimodal Video Understanding

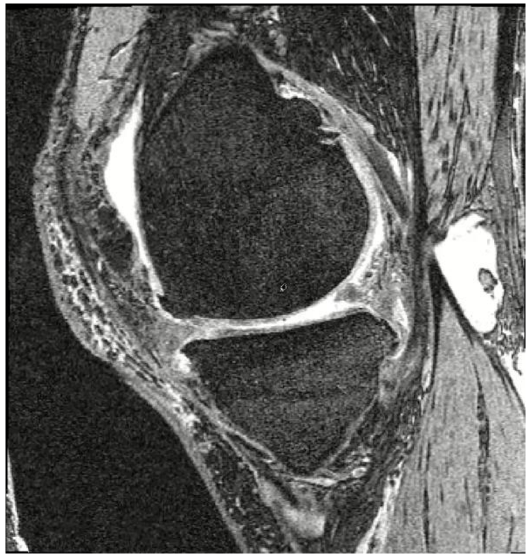


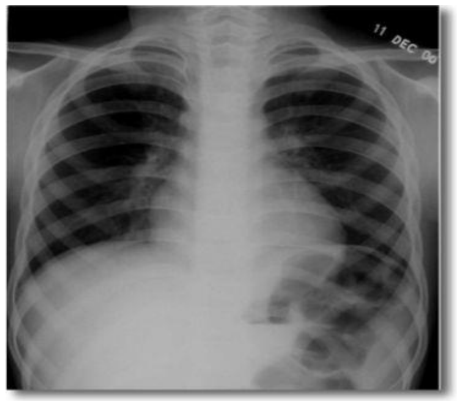
Visualization & Understanding

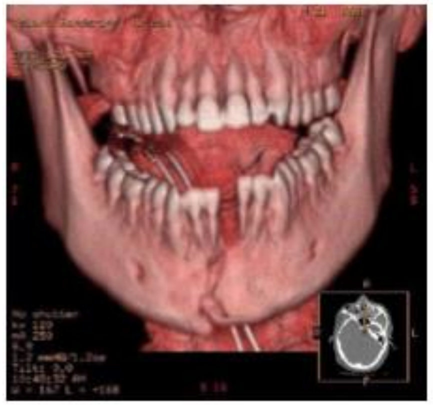
Self-driving Cars

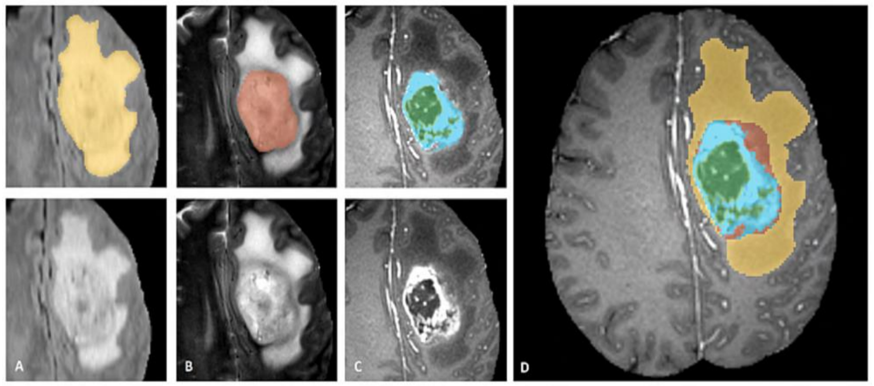
Medical Image Data is Everywhere



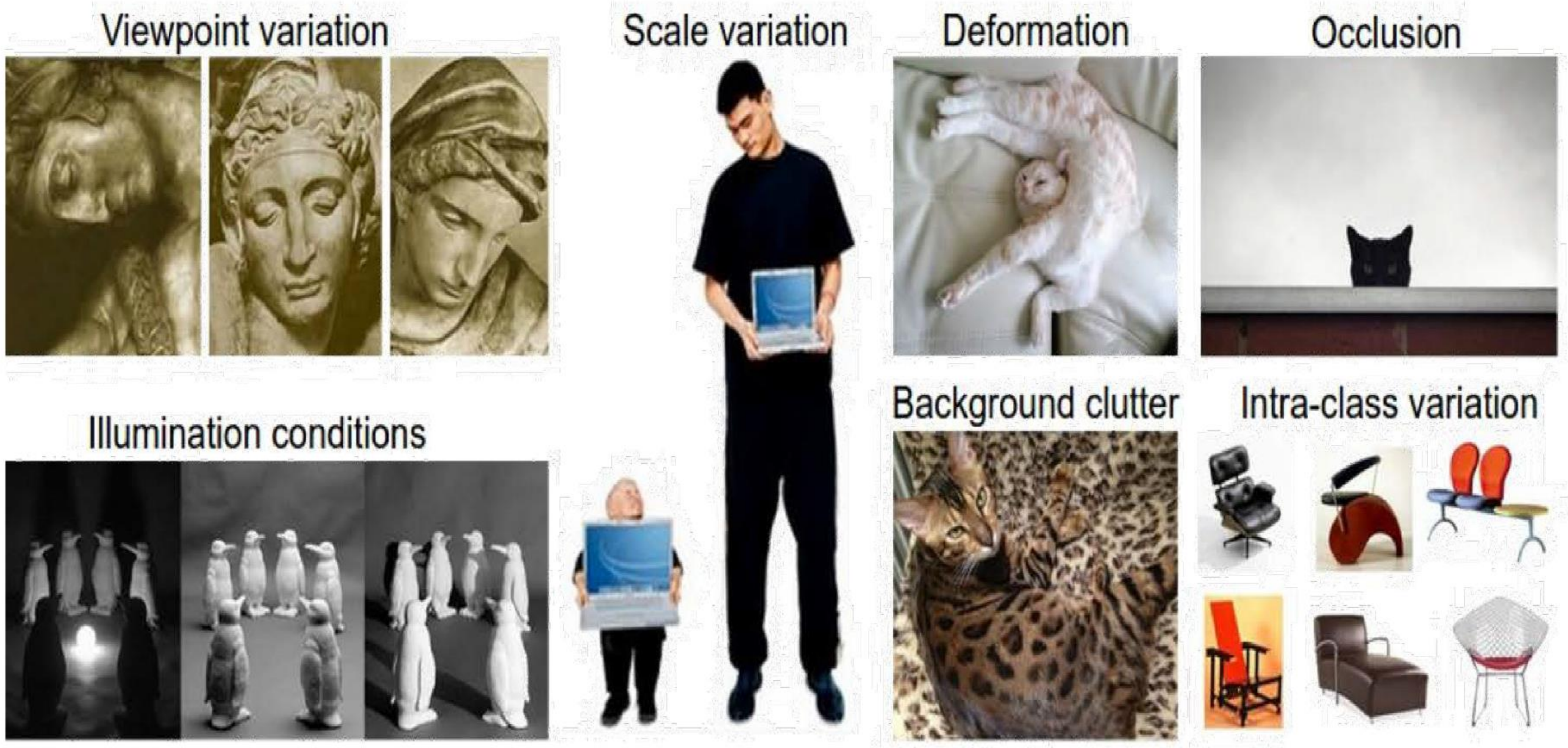








Scenario Challenges



High Dimensionality

- **Key Feature**:
  - Images are inherently high-dimensional data. For example, a standard image in classification tasks with a resolution of 224×224224×224 and 3 color channels (RGB) has 224×224×3=150,528224×224×3=150,528 input dimensions.
  - Each pixel represents a separate input feature, and the number of features grows quadratically with image resolution.
- **Challenge**:
  - Fully connected networks scale poorly with such high-dimensional data. For even a shallow network, the number of weights can exceed 150,5282150,5282 (~22 billion). This massive number of weights:
    - Increases the risk of overfitting, as more parameters require a proportional increase in training data.
    - Results in impractical memory and computational requirements, especially for larger images.
    - Slows down the training process significantly, making optimization difficult.
- **Real-World Implication**:
  - As image resolution increases (e.g., 512×512512×512 or beyond for high-definition images), the dimensionality becomes even more unmanageable for fully connected networks.
- **Solution**: CNNs reduce the number of parameters by using **shared weights (convolutional filters) and processing local regions of the image (kernels).** This drastically decreases memory requirements and computational complexity.

Spatial Relationships in Pixels

- **Key Feature**:
  - Nearby pixels in an image are statistically correlated and form local patterns or textures (e.g., edges, corners, and gradients). These local relationships are critical for understanding the content of an image.
  - For example, in an image of a cat, nearby pixels may collectively form the texture of fur or the shape of an ear.
- **Challenge**:
  - Fully connected networks ignore spatial relationships by treating all input pixels equally. They lack the notion of "locality" and process the relationship between each pixel and every other pixel, regardless of their proximity.
  - This lack of spatial awareness means that a fully connected network cannot naturally exploit the structural dependencies within an image.
  - If the pixels of an image are randomly permuted in the same way for both training and testing, a fully connected network can still learn, highlighting its disregard for spatial coherence.
- **Real-World Implication**:
  - Without spatial awareness, models become inefficient and require a larger number of neurons to learn even basic patterns.
- **Solution**:
  - CNNs address this by using **local receptive fields** to capture spatial relationships. Filters (kernels) process small, overlapping regions of an image, preserving spatial coherence and focusing on local patterns. This makes CNNs particularly effective for tasks like object detection and image segmentation.

Stability Under Geometric Transformations

- **Key Feature**:
  - Images maintain their interpretation under geometric transformations such as translation, rotation, scaling, or flipping. For example:
    - A tree remains recognizable as a tree even if shifted slightly to the left or rotated by a small angle.
    - Similarly, a flipped or resized image of a cat does not change its underlying identity.
  - This invariance is essential for real-world applications like autonomous driving or medical imaging, where objects may appear in various positions or orientations.
- **Challenge**:
  - Fully connected networks treat each pixel independently and do not account for geometric transformations. A simple translation (e.g., shifting an image to the left by a few pixels) alters every pixel in the input vector, forcing the network to relearn patterns for each possible position.
  - This redundancy results in inefficient learning and requires significantly more data to cover all potential transformations.
- **Real-World Implication**:
  - Models that lack invariance to transformations are less robust in real-world scenarios where objects appear in varying contexts.
- **Solution**:
  - CNNs inherently address this issue by leveraging **translation invariance through shared filters**. These filters recognize patterns (e.g., edges or textures) regardless of their position within the image.
  - **Data augmentation techniques**, such as randomly rotating, flipping, or cropping images during training, further improve the model's ability to handle transformations.

Additional Considerations

- **Noise in Images**:
  - Real-world images often contain noise (e.g., sensor artifacts, motion blur, or lighting variations). Fully connected networks struggle to differentiate between noise and meaningful patterns, further emphasizing the need for specialized architectures.
  - CNNs are more robust to noise due to their focus on local features rather than individual pixel values.
- **Scale and Hierarchy**:
  - Images often contain hierarchical features at multiple scales:
    - Low-level features: edges, corners.
    - Mid-level features: textures, patterns.
    - High-level features: objects or entire scenes.
  - Fully connected networks cannot naturally represent this hierarchy, while CNNs achieve this using multiple convolutional layers with increasing receptive fields.
- **Conclusion** The unique properties of unstructured image data pose significant challenges for fully connected networks. These challenges necessitate specialized architectures like CNNs, which **leverage shared weights, local receptive fields, and hierarchical feature extraction to process images efficiently**. Additionally, techniques like data augmentation and multi-scale analysis enhance the robustness of these models for real-world applications.

Content

Introduction to CNN

What Are CNNs?

CNNs are specialized deep learning architectures designed to process data with grid-like structures, such as images and videos. By leveraging the spatial structure of data, CNNs efficiently extract and learn hierarchical features, making them particularly well-suited for computer vision tasks like image classification, object detection, and segmentation.

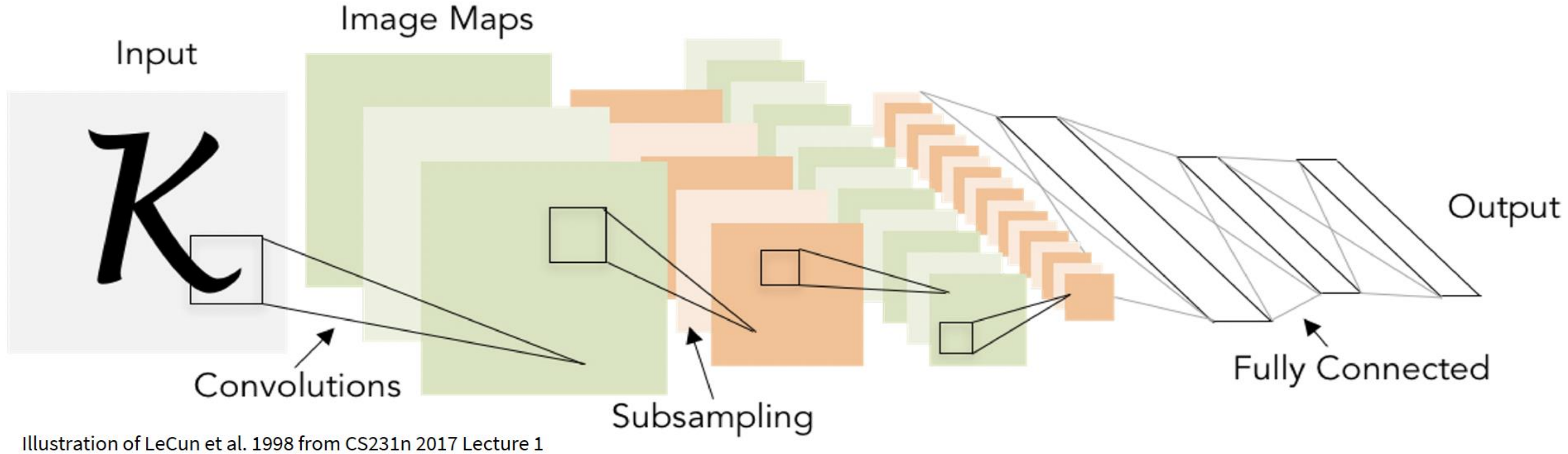


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

CNNs' applications

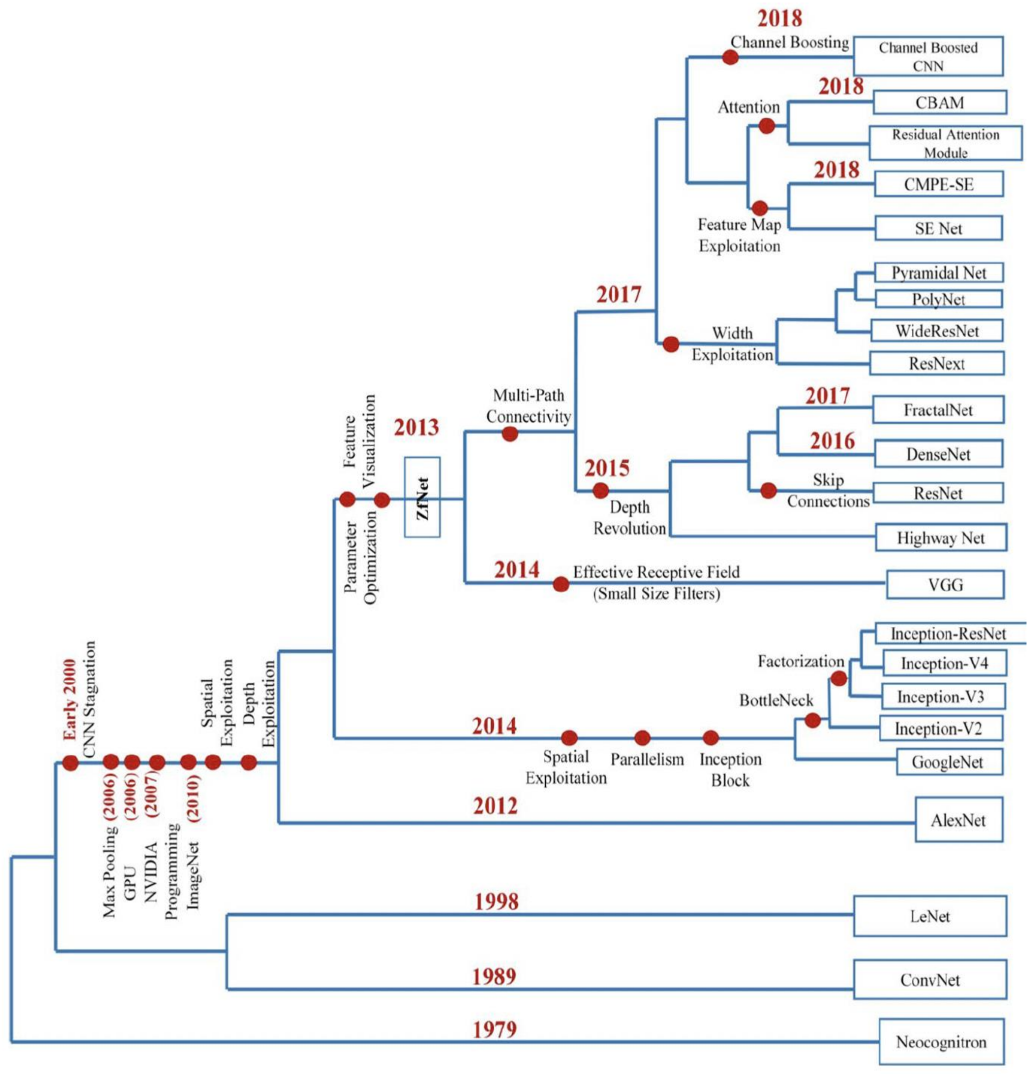
In **image and video processing**, they are widely used for tasks such as classification, object detection, segmentation, and face recognition.

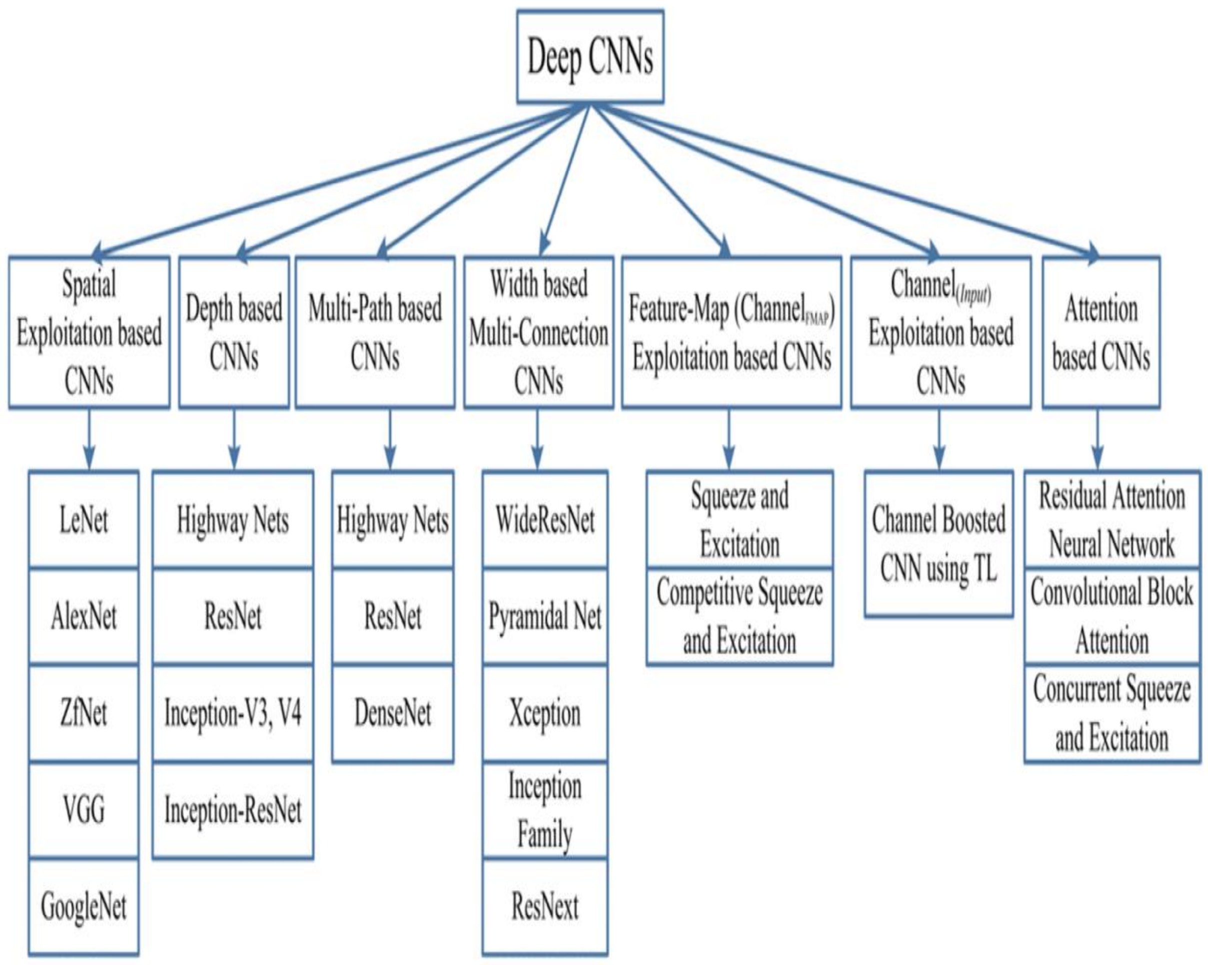
In **medical imaging**, CNNs assist in detecting tumors and anomalies in X-rays and CT scans.

In **natural language processing** (NLP), they process data as 1D inputs for tasks like sentence classification and text summarization.

In **autonomous driving,** they enable real-time object detection for pedestrians, vehicles, and road signs.

CNN History and Categories



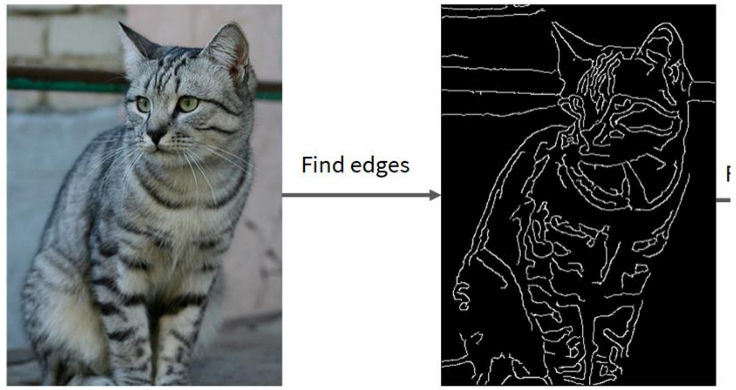


Deep CNNs
Spatial Exploitation based CNNs
Depth based CNNs
Multi-Path based CNNs
Width based Multi-Connection CNNs
Feature-Map (Channel$_{FMAP}$) Exploitation based CNNs
Channel$_{(Input)}$ Exploitation based CNNs
Attention based CNNs
LeNet
AlexNet
ZfNet
VGG
GoogleNet
Highway Nets
ResNet
Inception-V3, V4
Inception-ResNet
Highway Nets
ResNet
DenseNet
WideResNet
Pyramidal Net
Xception
Inception Family
ResNext
Squeeze and Excitation
Competitive Squeeze and Excitation
Channel Boosted CNN using TL
Residual Attention Neural Network
Convolutional Block Attention
Concurrent Squeeze and Excitation

Inspiration Behind CNN

CNNs were inspired by the layered architecture of the human visual cortex.

- **Hierarchical architecture:** Both CNNs and the visual cortex have a hierarchical structure, with simple features extracted in early layers and more complex features built up in deeper layers. This allows increasingly sophisticated representations of visual inputs.
- **Local connectivity:** Neurons in the visual cortex only connect to a local region of the input, not the entire visual field. Similarly, the neurons in a CNN layer are only connected to a local region of the input volume through the convolution operation. This local connectivity enables efficiency.



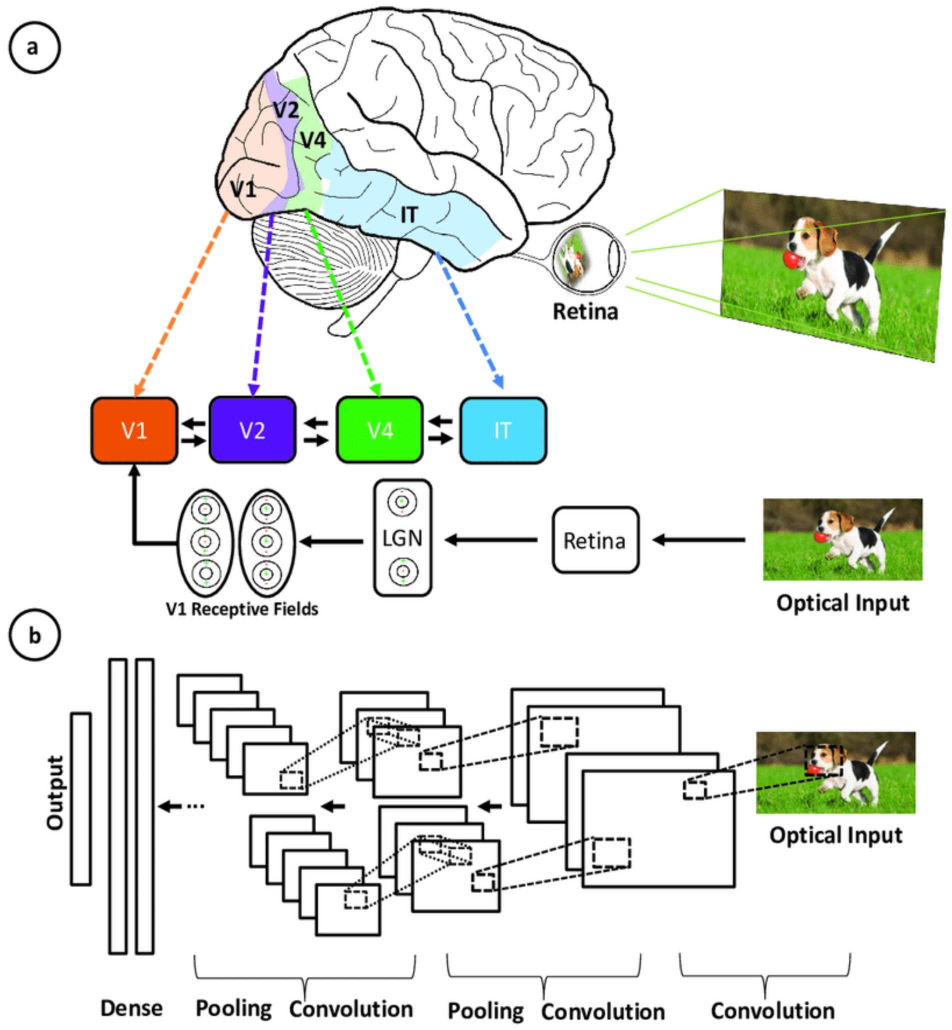
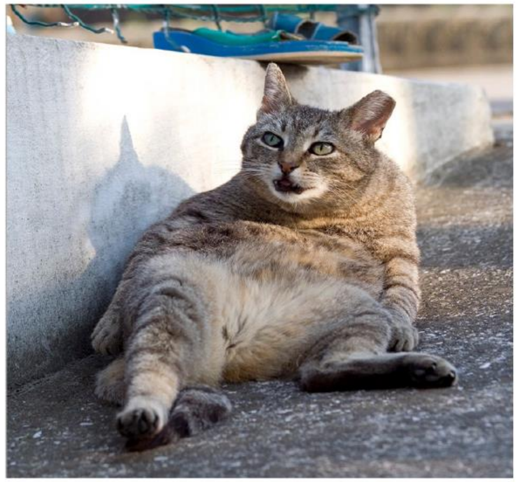
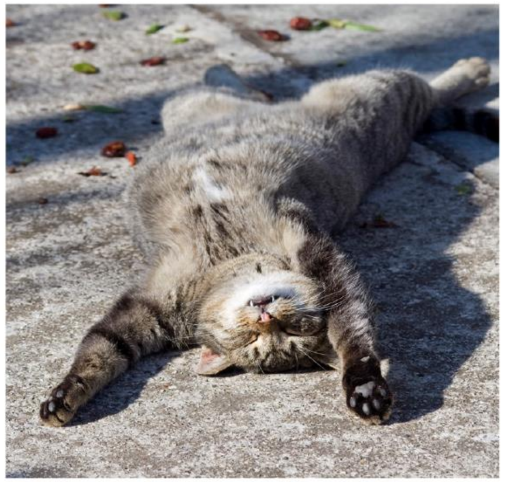


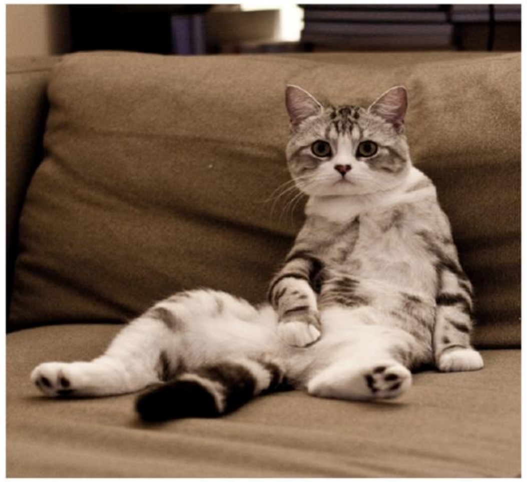
Illustration of the correspondence between the areas associated with the primary visual cortex and the layers in a convolutional neural network (source)

Inspiration Behind CNN


This image by Umberto Salvagnin is licensed under CC-BY 2.0


This image by Umberto Salvagnin is licensed under CC-BY 2.0

This image by sare bear is licensed under CC-BY 2.0


This image by Tom Thai is licensed under CC-BY 2.0

- **Translation invariance**:

Similar to visual cortex neurons can detect features regardless of their location in the visual field, pooling layers in a CNN provide a degree of translation invariance by summarizing local features.

- **Multiple feature maps:**

At each stage of visual processing, there are many different feature maps extracted. CNNs mimic this through multiple filter maps in each convolution layer.

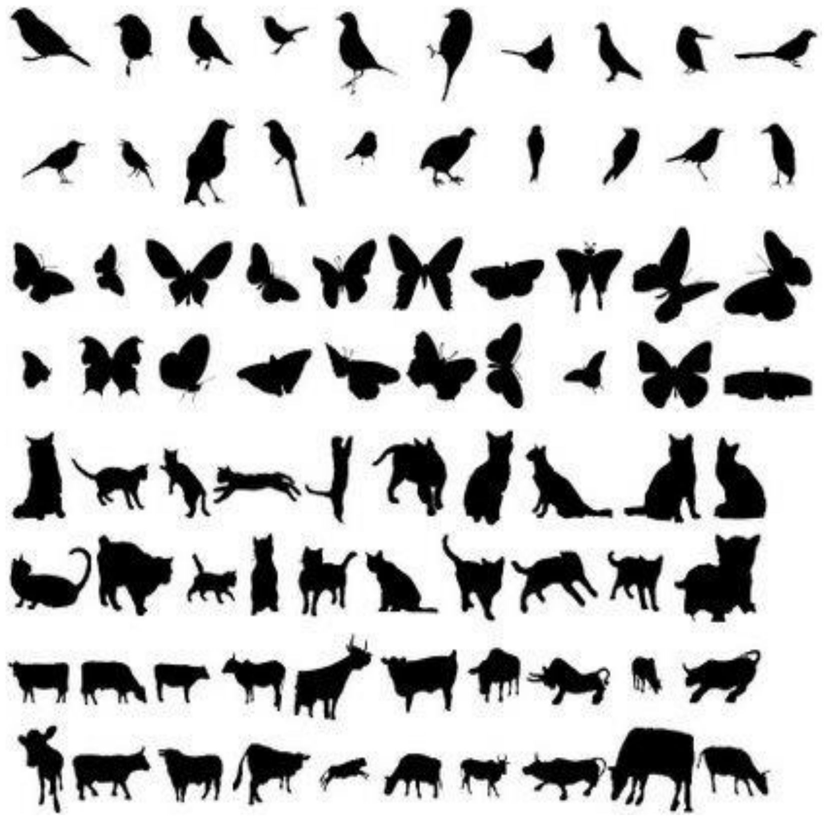
- **Non-linearity:**

Neurons in the visual cortex exhibit non-linear response properties. CNNs achieve non-linearity through activation functions like ReLU applied after each convolution.

Understanding Invariance in CNNs

- **What is Invariance?**
  - Invariance refers to properties or quantities that remain unchanged under certain transformations or operations.
  - In the context of deep learning, invariance refers to a model's ability to recognize patterns regardless of certain transformations (e.g. Translation (shifting), Rotation, Scaling), as the model's output remains unchanged under certain transformations of the input.
- **Why is Invariance Important?**
  - Aids in simplifying complex problems by focusing on constant factors.
  - Real-world objects may appear in different positions, orientations, or scales.
  - Models should recognize objects regardless of these variations.
- **Limitations of Fully Connected Networks (MLPs)**
  - Do not inherently handle spatial hierarchies in data.
  - Lack of invariance to translations and other transformations.

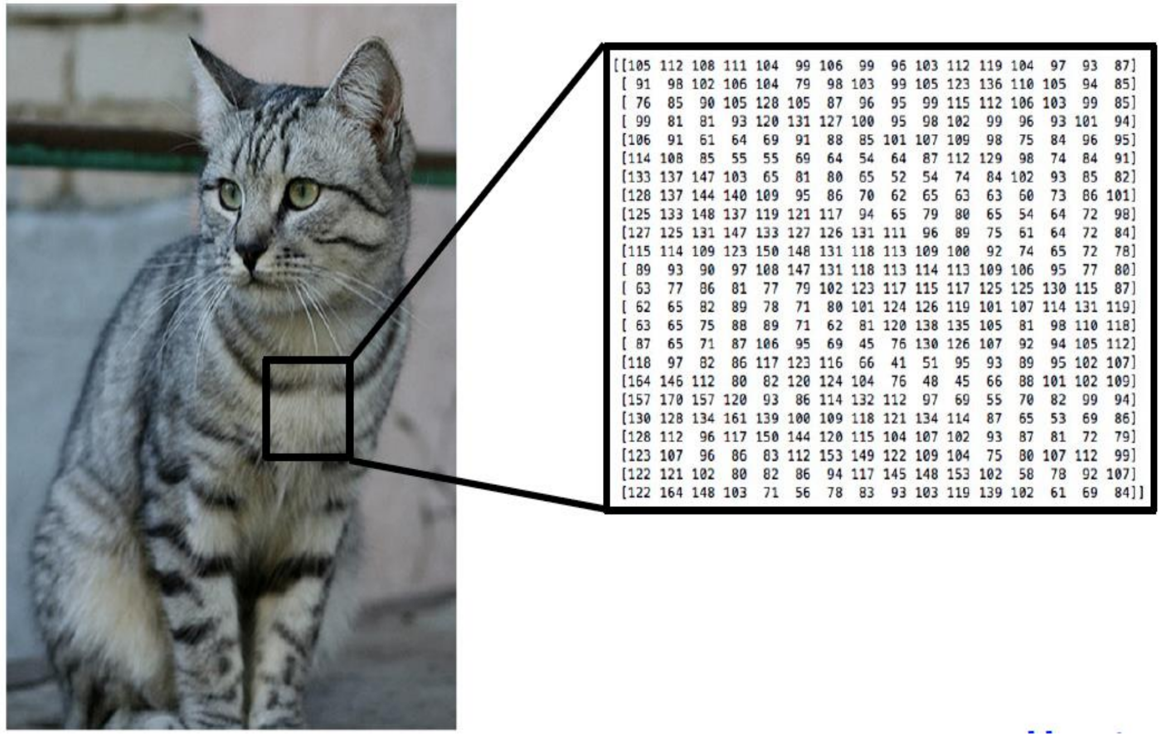
Shape Classes



http://sites.google.com/site/xiangbai/try-large.jpg
http://sites.google.com/site/xiangbai/animaldataset

How Invariance Inspires CNN

Invariance is a guiding principle in CNN architecture, enabling robust and efficient processing of complex visual data. By leveraging translation invariance, spatial hierarchies, weight sharing, and robustness to transformations, CNNs achieve exceptional performance in tasks. This combination of mathematical rigor and biological inspiration has made CNNs a cornerstone of modern computer vision.



```
[[105 112 108 111 104  99 106  99  96 103 112 119 104  97  93  87]
 [ 91  98 102 106 104  79  98 103  99 105 123 136 110 105  94  85]
 [ 76  85  90 105 128 105  87  96  95  99 115 112 106 103  99  85]
 [ 99  81  81  93 120 131 127 100  95  98 102  99  96  93 101  94]
 [106  91  61  64  69  91  88  85 101 107 109  98  75  84  96  95]
 [114 108  85  55  55  69  64  54  64  87 112 129  98  74  84  91]
 [133 137 147 103  65  81  80  65  52  54  74  84 102  93  85  82]
 [128 137 144 140 109  95  86  70  62  65  63  63  60  73  86 101]
 [125 133 148 137 119 121 117  94  65  79  80  65  54  64  72  98]
 [127 125 131 147 133 127 126 131 111  96  89  75  61  64  72  84]
 [115 114 109 123 150 148 131 118 113 109 100  92  74  65  72  78]
 [ 89  93  90  97 108 147 131 118 113 114 113 109 106  95  77  80]
 [ 63  77  86  81  77  79 102 123 117 115 117 125 125 130 115  87]
 [ 62  65  82  89  78  71  80 101 124 126 119 101 107 114 131 119]
 [ 63  65  75  88  89  71  62  81 120 138 135 105  81  98 110 118]
 [ 87  65  71  87 106  95  69  45  76 130 126 107  92  94 105 112]
 [118  97  82  86 117 123 116  66  41  51  95  93  89  95 102 107]
 [164 146 112  80  82 120 124 104  76  48  45  66  88 101 102 109]
 [157 170 157 120  93  86 114 132 112  97  69  55  70  82  99  94]
 [130 128 134 161 139 100 109 118 121 134 114  87  65  53  69  86]
 [128 112  96 117 150 144 120 115 104 107 102  93  87  81  72  79]
 [123 107  96  86  83 112 153 149 122 109 104  75  80 107 112  99]
 [122 121 102  80  82  86  94 117 145 148 153 102  58  78  92 107]
 [122 164 148 103  71  56  78  83  93 103 119 139 102  61  69  84]]
```

- **Convolutional Layers and Weight Sharing**
  - Convolutional layers apply the same filter across different spatial locations.
  - Weight sharing reduces the number of parameters and captures local patterns.
- **Translation Invariance**
  - Convolutions allow the detection of features regardless of their position.
  - Pooling layers further enhance invariance by summarizing nearby outputs.
- **Benefits for Visual Tasks**
  - Efficiently recognize objects in varied contexts.
  - Improve generalization by focusing on relevant features.

Key Components of CNN

- Convolutional layers
- Rectified Linear Unit (ReLU)
- Pooling layers
- Fully connected layers

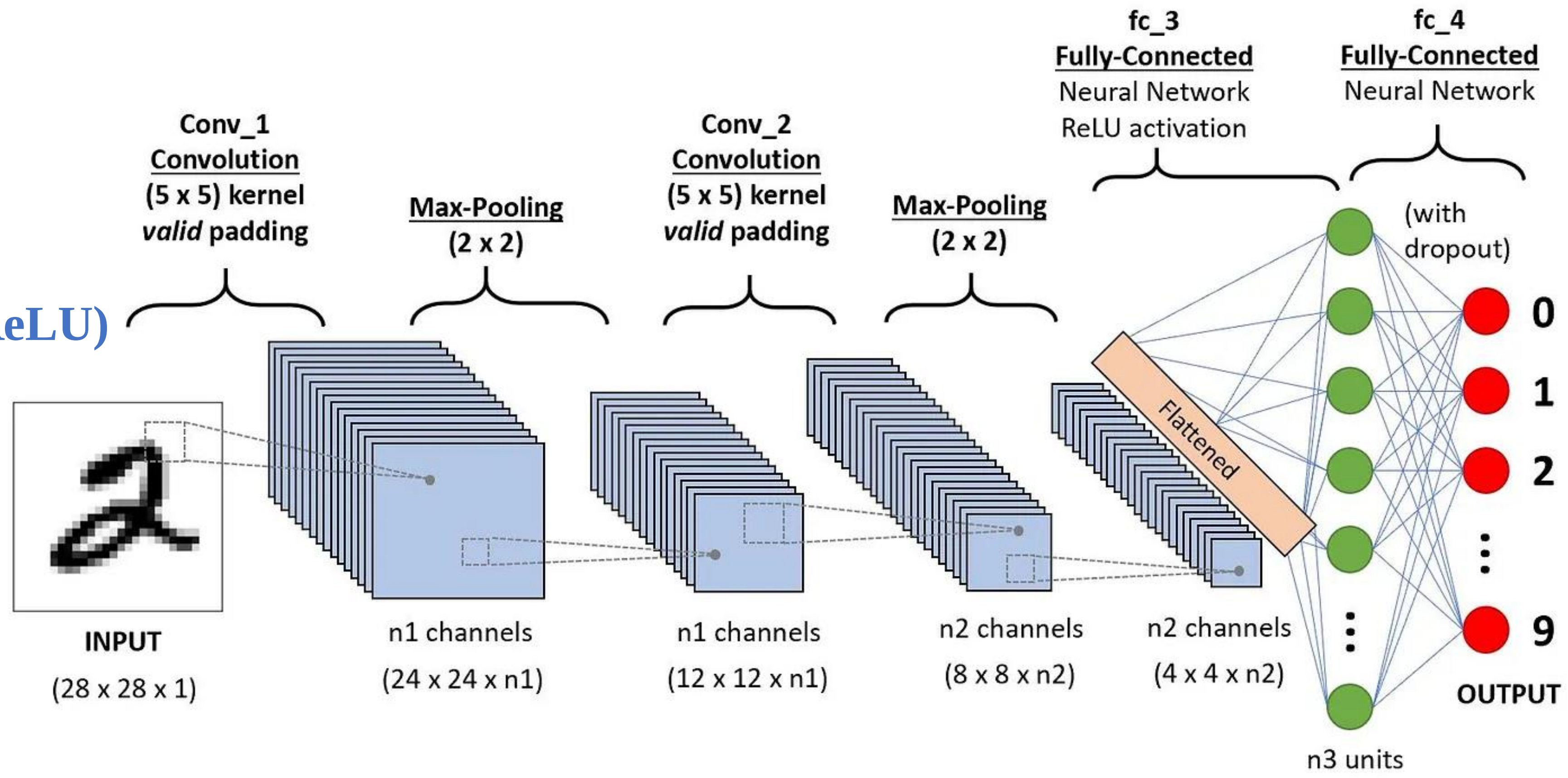
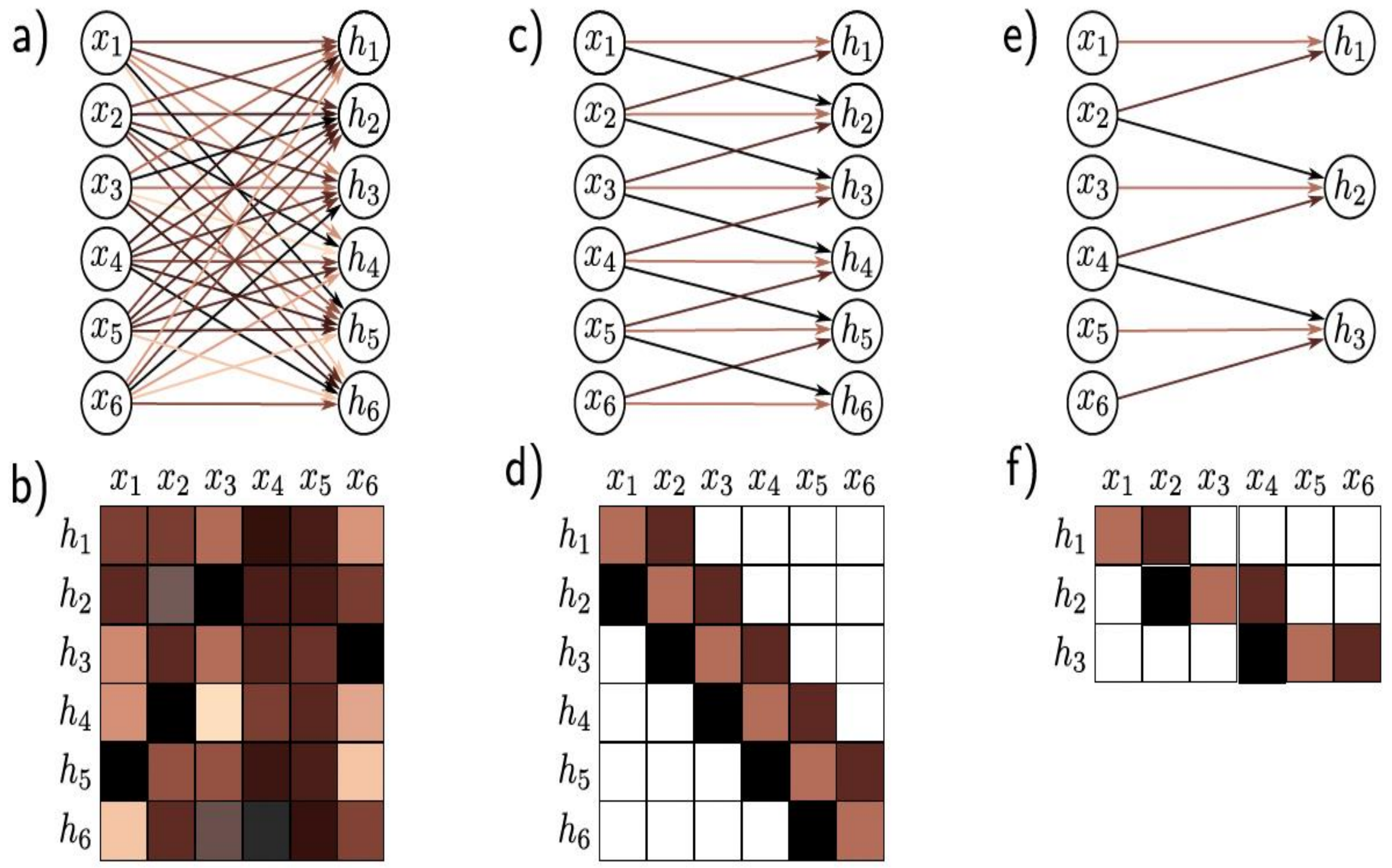


Illustration of architecture of CNNs applied to digit recognition (source)

Feature Extraction Using Convolution

- **Input, kernel, and output**
- **Fully Connected Networks**
  - “fully connect” all the hidden units to all the input units. Only computationally feasible to learn features on the entire image for relatively small images.
  - order of 10^6 parameters to learn for 96x96 images. The feedforward and backpropagation computations would also be about 100 times slower, compared to 28x28 images.
- **Locally Connected Networks**


a)
b)
c)
d)
e)
f)
x_1 x_2 x_3 x_4 x_5 x_6
h_1 h_2 h_3 h_4 h_5 h_6

Feature Extraction Using Convolution

- **Input, kernel, and output (right figure)**
- **Fully Connected Networks**
- **Locally Connected Networks**
    - A simple solution to this problem is to limit connections between hidden and input units, allowing each hidden unit to connect to only a small subset of input units, such as a contiguous region of pixels. For other data types different than images like audio, hidden units can be connected to specific time spans. This concept of local connections is inspired by the visual cortex, where neurons respond to stimuli in specific locations.

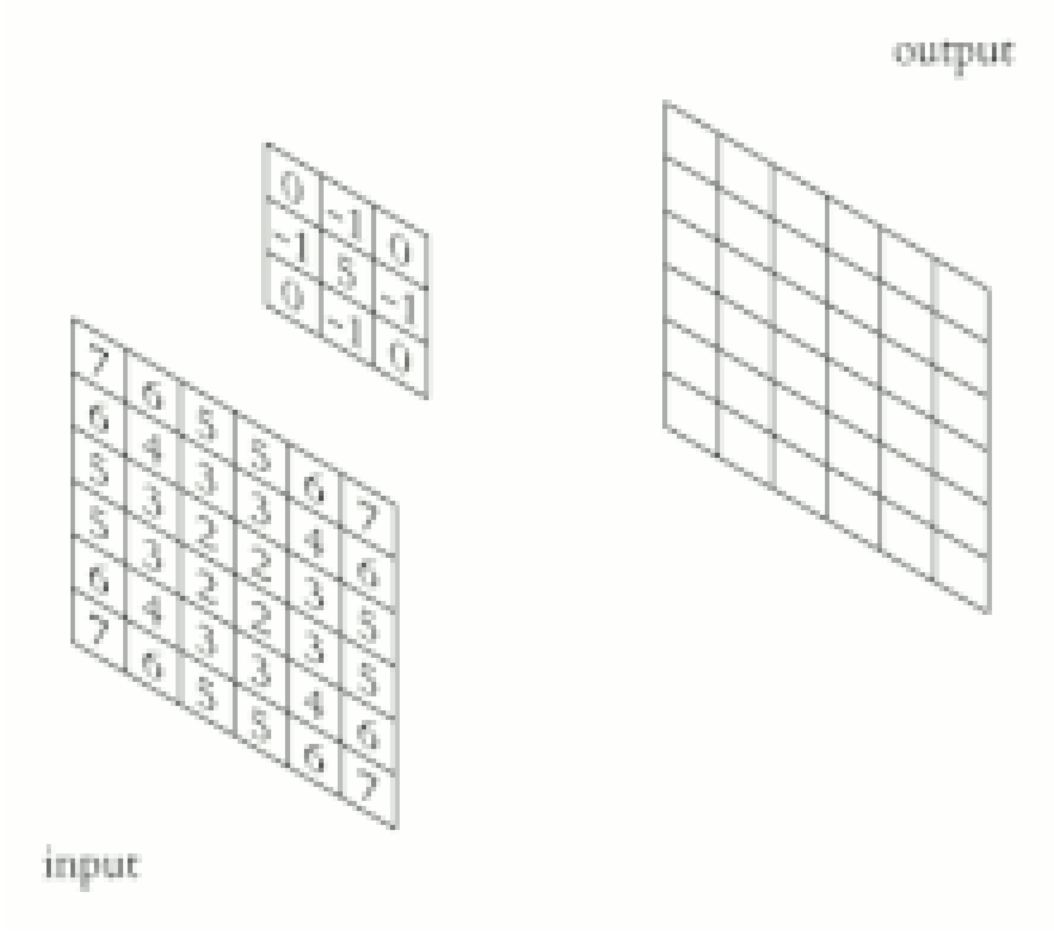


Illustration of Discrete 2D Convolution (source)

Understanding the Convolution Operation

What is convolution?

Mathematically, Convolution is defined as $f, g\colon \mathbb{R}^n \to \mathbb{R}$:

$$(f * g)(\boldsymbol{x}) = \int f(\boldsymbol{z})g(\boldsymbol{x} - \boldsymbol{z})dz$$

Whenever we have discrete objects, the integral turns into a sum. For instance, in CNN, we used discrete convolution for vectors from the set of square-summable infinite-dimensional vectors defined as:

$$(f * g)(i) = \sum_a f(i)g(i - a)$$

For two-dimensional tensors, we have a corresponding sum with (a,b) for f (i-a,j-b) for g, respectively:

$$(f * g)(i, j) = \sum_a \sum_b f(\mathrm{a}, \mathrm{b})g(i - a, j - b)$$

Exercises

1. Audio data is often represented as a one-dimensional sequence.
    1. When might you want to impose locality and translation invariance for audio?
    2. Derive the convolution operations for audio.
    3. Can you treat audio using the same tools as computer vision? Hint: use the spectrogram.
2. Why might translation invariance not be a good idea after all? Give an example.
3. Do you think that convolutional layers might also be applicable for text data? Which problems might you encounter with language?
4. What happens with convolutions when an object is at the boundary of an image?
5. Prove that the convolution is symmetric, i.e., $f * g = g * f$

Understanding the Convolution Operation

Why convolution works for images?

Natural images have the property of being "'stationary"', meaning that the statistics of one part of the image are the same as any other part.

Formally, given some large r×c images x_{large}, we first train a sparse autoencoder on small a×b patches x_{small} sampled from these images, learning k features $f = \sigma(W^{(1)}X_{small} + b^{(1)})$ (where σ is the sigmoid function), given by the weights $W^{(1)}$and biases $b^{(1)}$ from the visible units to the hidden units. For every a×b patch x_s in the large image, we compute

$f_s = \sigma(W^{(1)}x_s + b^{(1)})$, giving us $\mathbf{f_{convolved}}$,

a k × (r−a+1) × (c−b+1) array of convolved features.

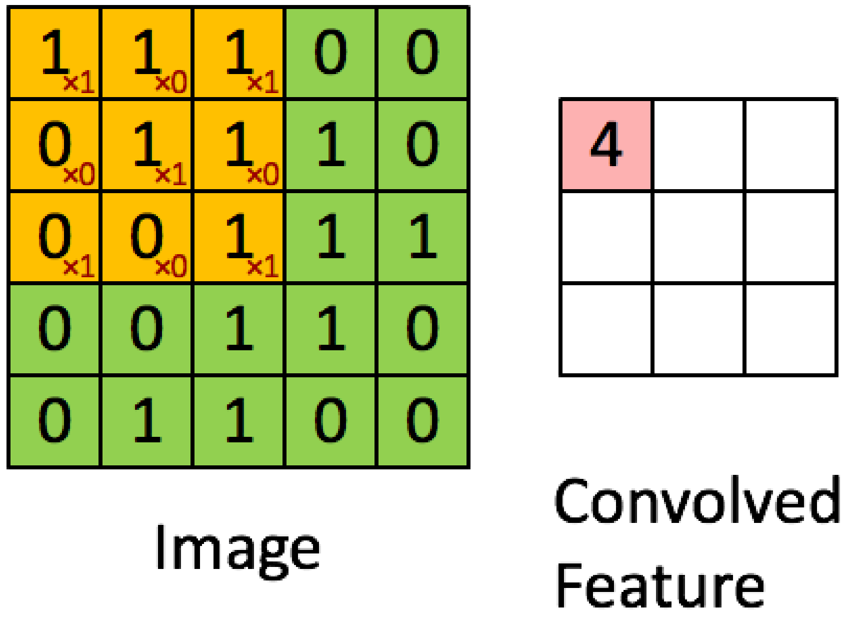
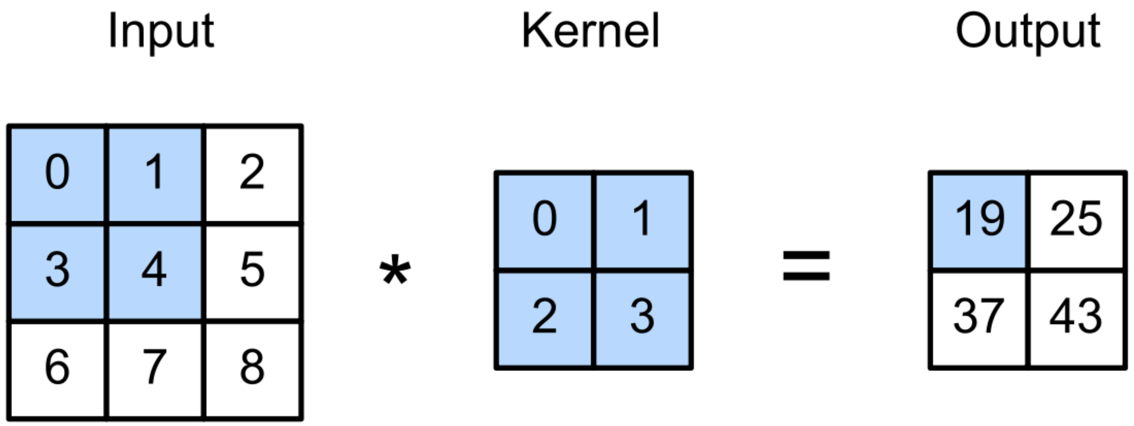


Illustration of Discrete 2D Convolution (source)

Cross-Correlation Operation

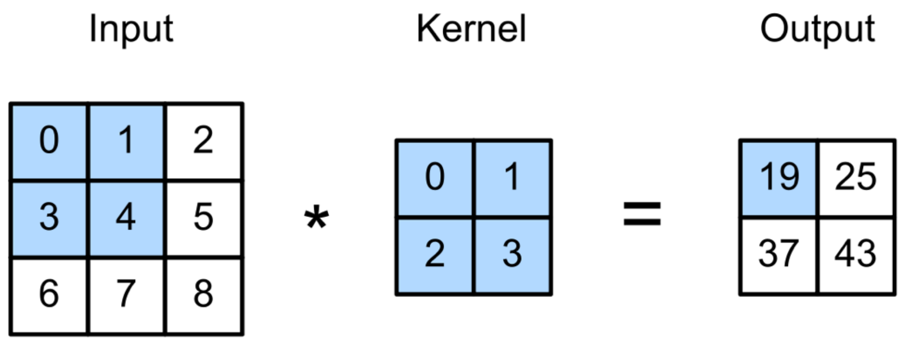
In practice, convolution operations in CNN can be more accurately described as cross-correlations. In each convolution layer, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation.

Consider example:



Cross-Correlation Operation

Consider example:



In the two-dimensional cross-correlation operation, we begin with the convolution window positioned at the upper-left corner of the input tensor and slide it across the input tensor, both from left to right and top to bottom. When the convolution window slides to a certain position, the input subtensor contained in that window and the kernel tensor are multiplied elementwise and the resulting tensor is summed up yielding a single scalar value. This result gives the value of the output tensor at the corresponding location.

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$
$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25$$
$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37$$
$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$

Cross-Correlation Operation

Note that along each axis, the output size is slightly smaller than the input size. Because the kernel has width and height greater than 1, we can only properly compute the cross-correlation for locations where the kernel fits wholly within the image.

Exercise:

Show that if convolution kernel size = 0, the convolution kernel implements an MLP independently for each set of channels. (Lin, M., Chen, Q., & Yan, S. (2013). Network in network. ArXiv:1312.4400.)

```
import torch

def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1,
X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] *
K).sum()
    return Y
```

Convolutional Layers

A convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output. The two parameters of a convolutional layer are the kernel and the scalar bias. When training models based on convolutional layers, we typically initialize the kernels randomly.

We can implement a two-dimensional convolutional layer based on the corr2d function defined above. In the __init__ constructor method, we declare weight and bias as the two model parameters. The forward propagation method calls the corr2d function and adds the bias.

```
import torch

class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight =
nn.Parameter(torch.rand(kernel_size))
        self.bias =
nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) +
self.bias
```

Illustration of Discrete 2D Convolution (source)

Padding, Stride, and Pooling

- **Padding**
  - Zero-padding and why it's necessary (The pixels at the corner in the previous images are less counted than those in the middle)
  - How padding affects the dimensions of the output

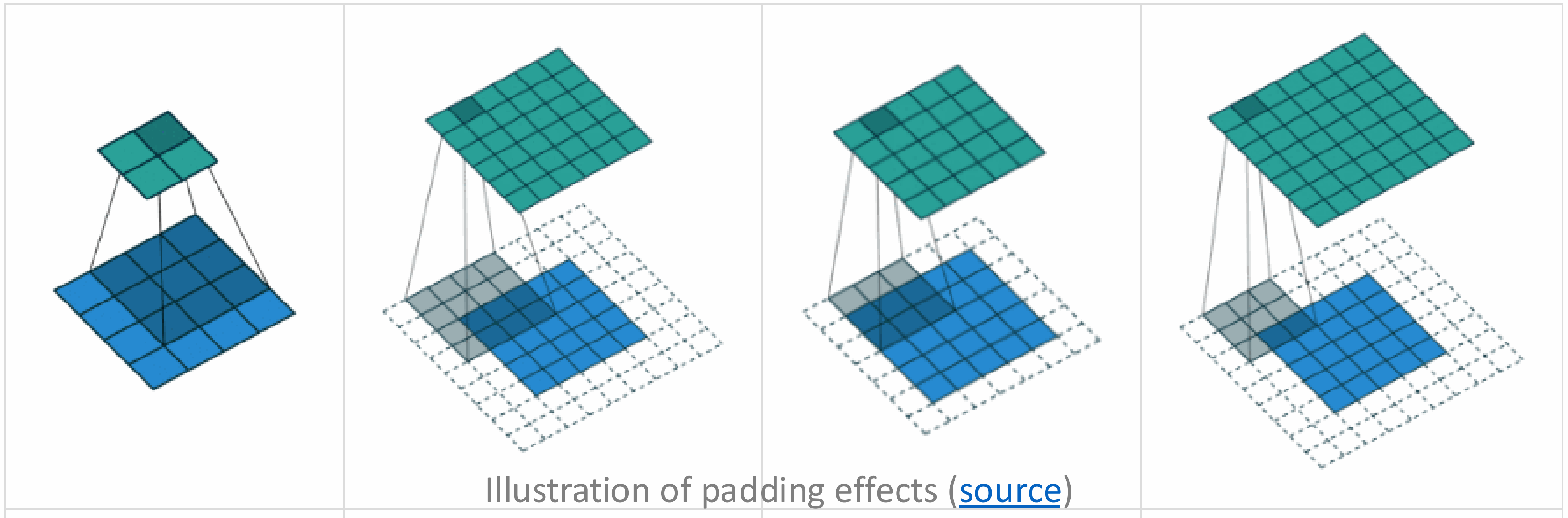
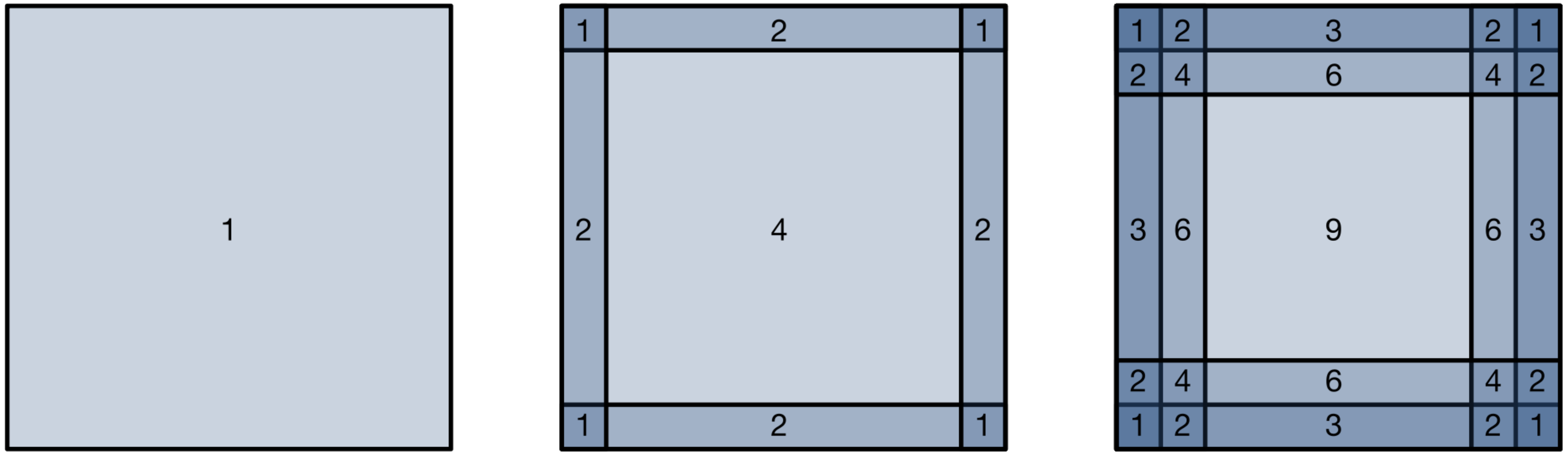


Illustration of padding effects (source)

Padding

One tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. The following figure depicts the pixel utilization as a function of the convolution kernel size and the position within the image.
We can see that the pixels in the corners are hardly used at all.

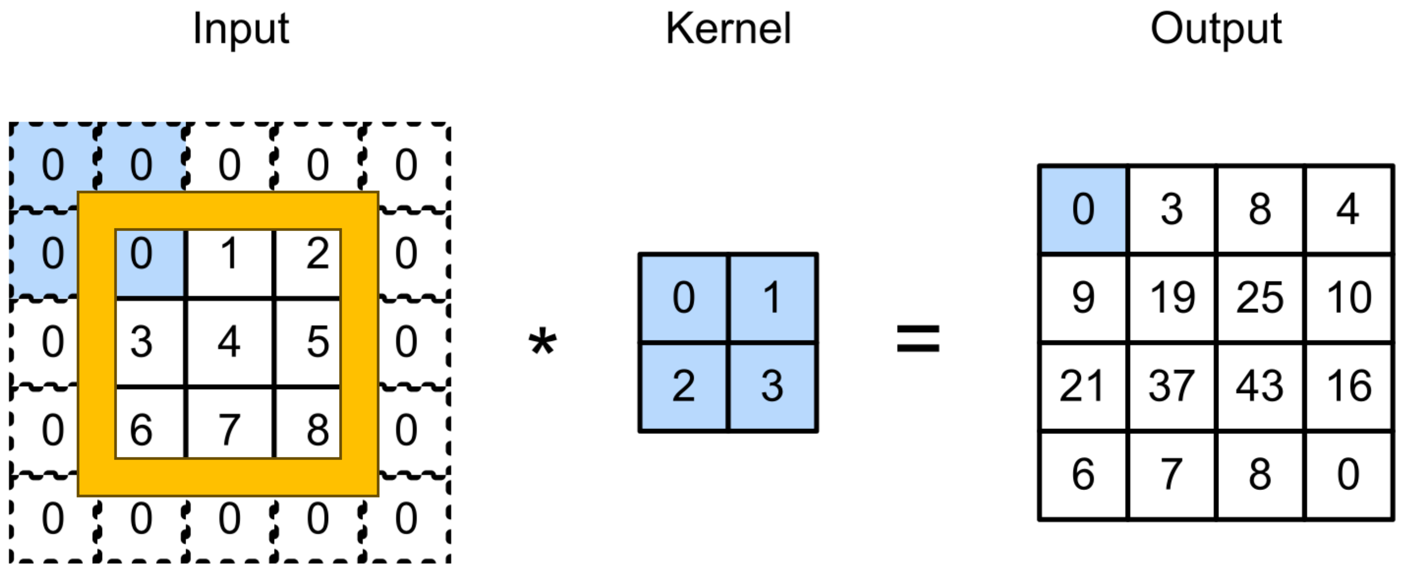


Pixel utilization for convolutions of 1x1, 2x2, and 3x3 respectively.

Padding

One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero.

Example on padding 3x3 input to 5x5 matrix:



Padding

In general, if we add a total of p_h rows of padding (roughly half on top and half on bottom) and a total of p_w columns of padding (roughly half on the left and half on the right), the height and width of the output will increase by p_h and p_w, respectively.

In many cases, we will want to set $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the input and output the same height and width.

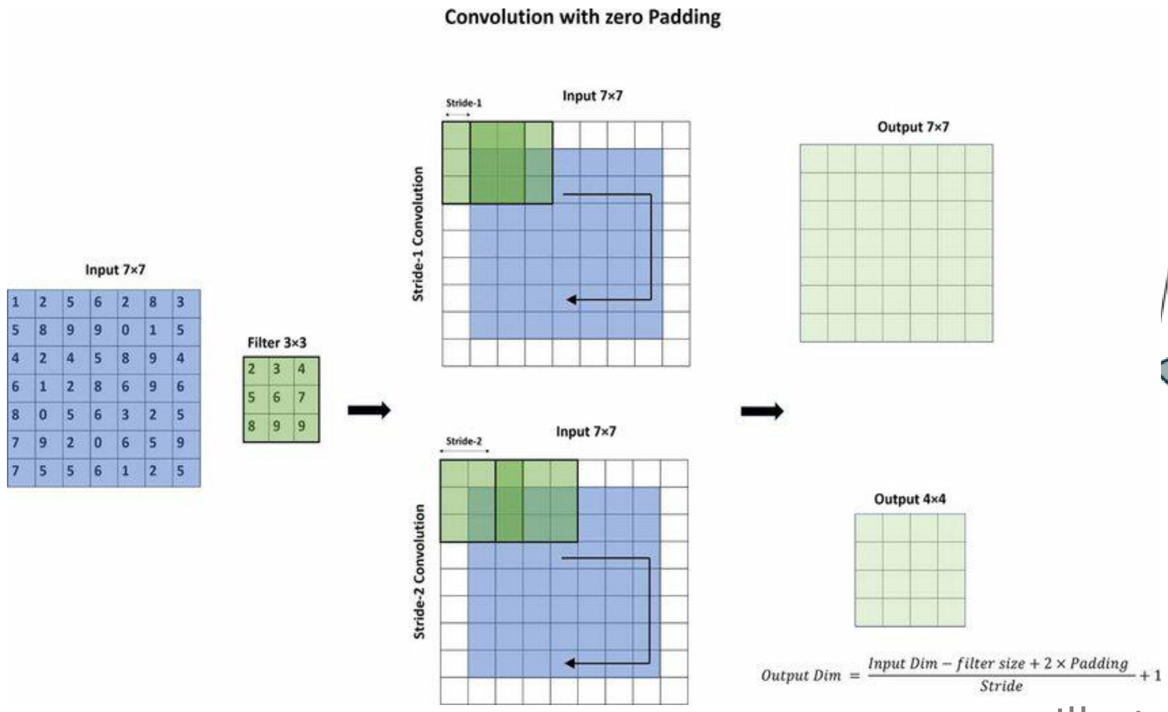
Example code to create a two-dimensional convolutional layer with a height and width of 3 and apply 1 pixel of padding on all sides.

```
from torch import nn
# We define a helper function to calculate convolutions. It initializes the
# convolutional layer weights and performs corresponding dimensionality
# elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    # (1, 1) indicates that batch size and the number of channels are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    return Y.reshape(Y.shape[2:])

# 1 row and column is padded on either side, so a total of 2 rows or columns are added
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

Padding, Stride, and Pooling

Stride

- Example with stride of 1 vs. 2



$$\text{Output Dim} = \frac{\text{Input Dim} - \text{filter size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

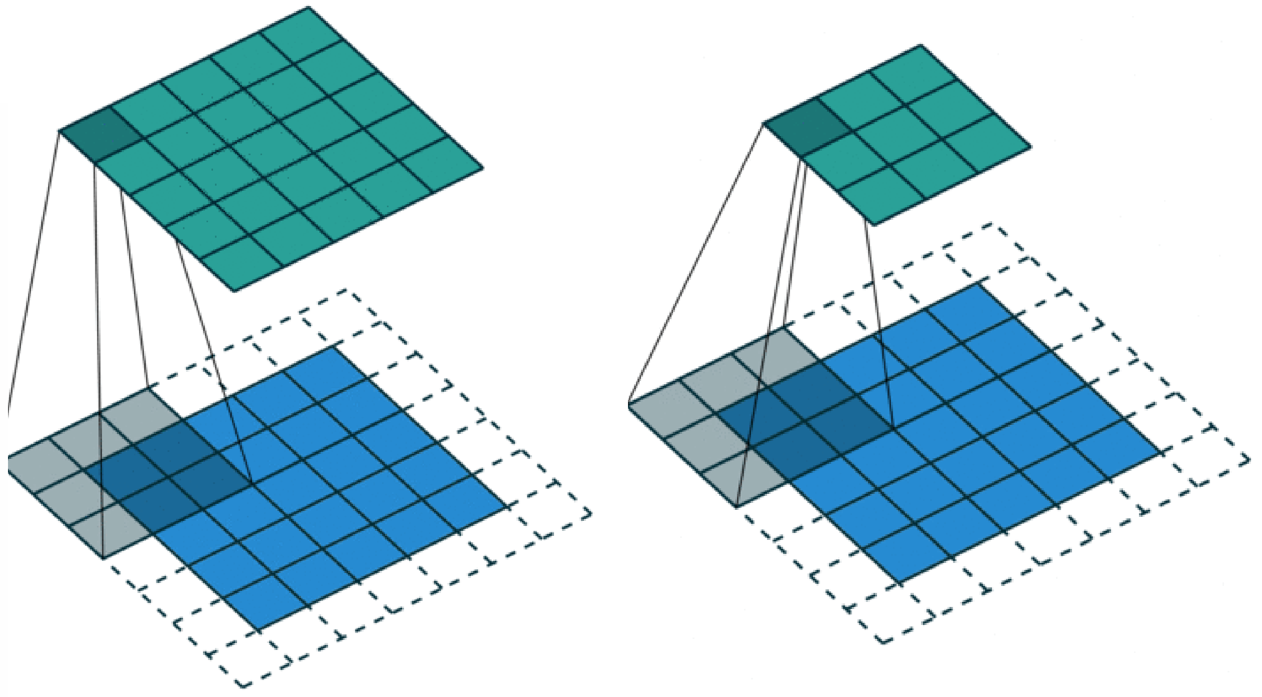


Illustration Convolution Operation with Stride Length = 1 Vs 2 (source)

Stride

We refer to the number of rows and columns traversed per slide as **stride**.

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\left\lfloor \frac{n_h - k_h + p_h + s_h}{s_h} \right\rfloor \times \left\lfloor \frac{n_w - k_w + p_w + s_w}{s_w} \right\rfloor.$$

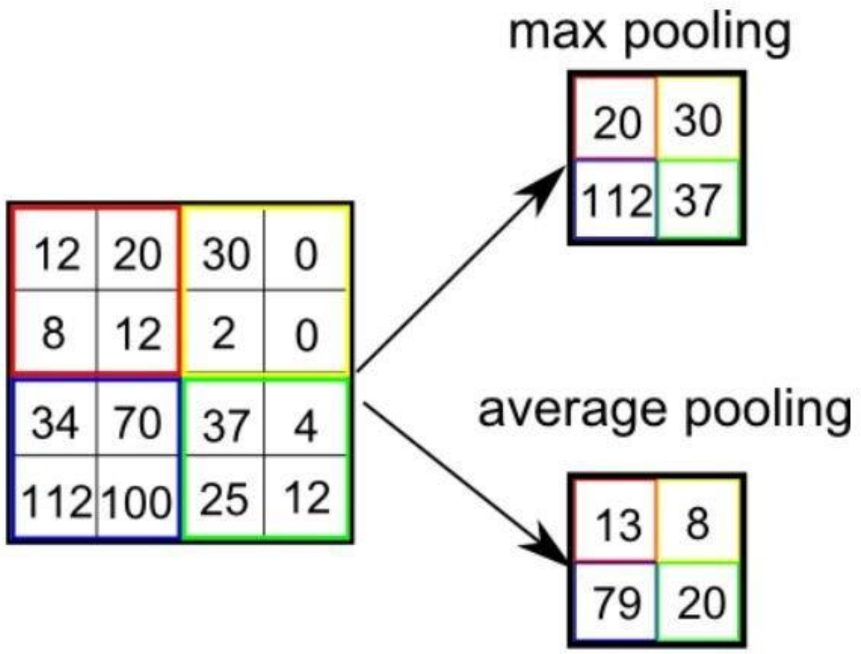
If we set $p_h = k_h - 1$ and $p_w = k_w - 1$, this can be simplified as

$$\left\lfloor \frac{n_h + s_h - 1}{s_h} \right\rfloor \times \left\lfloor \frac{n_w + s_w - 1}{s_w} \right\rfloor.$$

```
from torch import nn

# Using same example 8x8 matrix X
# If we set the strides on both the height
and width to 2, thus halving the input
height and width.
conv2d = nn.LazyConv2d(1, kernel_size=3,
padding=1, stride=2)
comp_conv2d(conv2d, X).shape
# torch.Size([4, 4])


# A slightly more complicated example.
conv2d = nn.LazyConv2d(1, kernel_size=(3,
5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
# torch.Size([2, 2])
```

Padding, Stride, and Pooling

- **Pooling**
  - Types: Max pooling, average pooling
  - Role in reducing dimensionality
  - Example: Pooling on an image



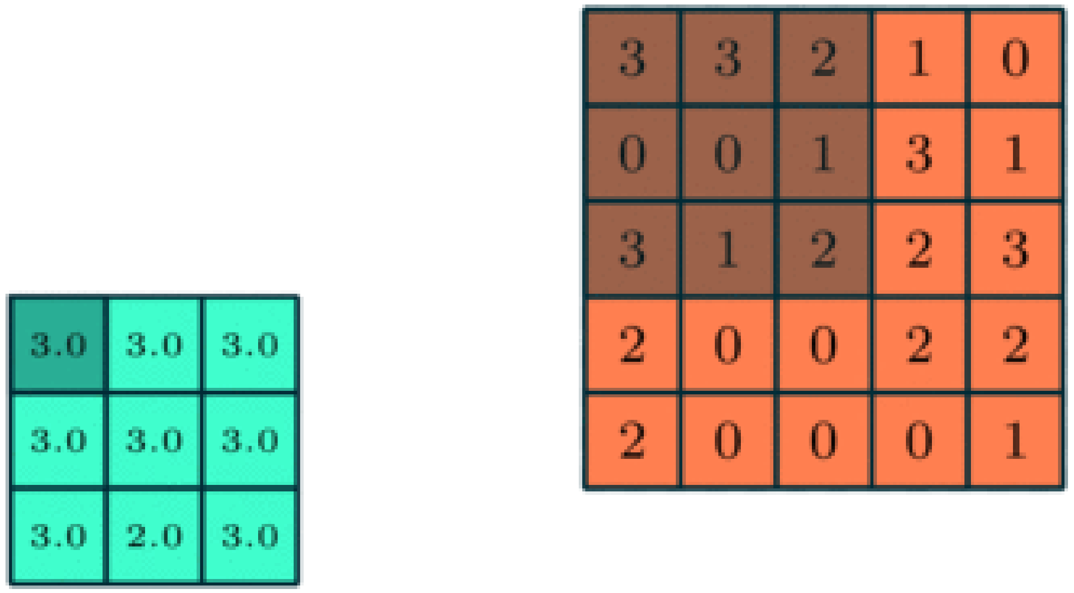


Illustration of 3x3 pooling over 5x5 convolved feature (source)

Pooling

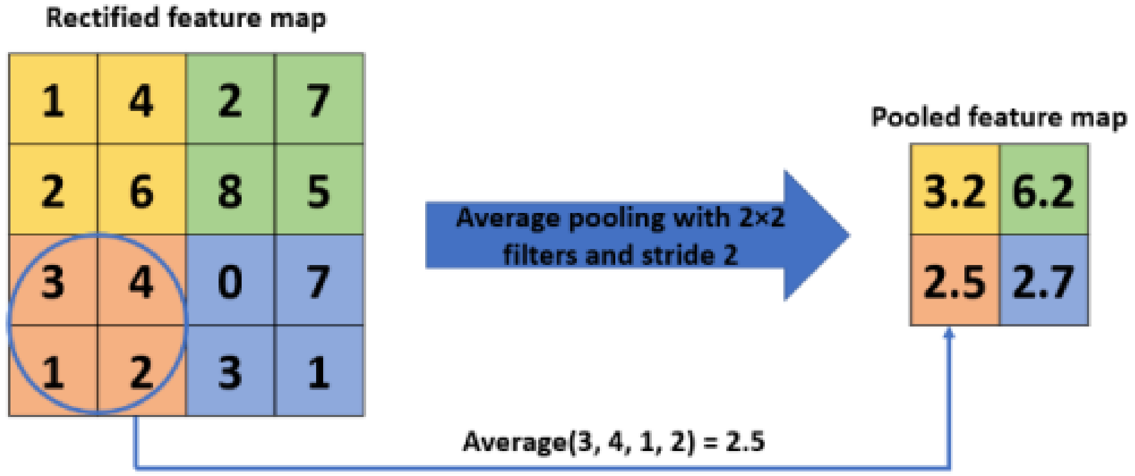
When detecting lower-level features, such as edges, we often want our representations to be somewhat invariant to translation. For instance, if we take the image X with a sharp delineation between black and white and shift the whole image by one pixel to the right, i.e., Z[i, j] = X[i, j + 1], then the output for the new image Z might be vastly different. The edge will have shifted by one pixel.

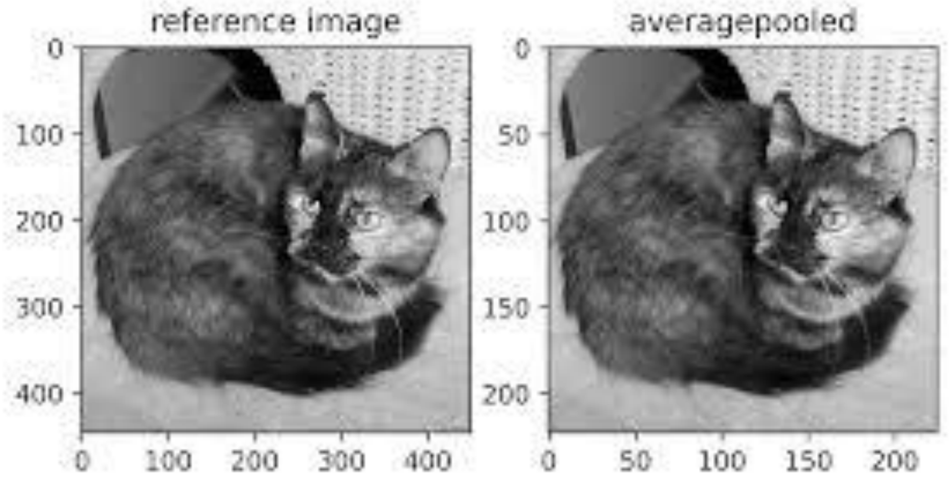
Pooling layers are introduced to serve the dual purposes of **mitigating the sensitivity of convolutional layers to location** and of **spatially downsampling representations**.

Unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters. Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called **maximum pooling** (max-pooling) and **average pooling**, respectively.

Average Pooling

Average pooling is essentially as old as CNNs. The idea is akin to downsampling an image. Rather than just taking the value of every second (or third) pixel for the lower resolution image, we can average over adjacent pixels to obtain an image with better signal-to-noise ratio since we are combining the information from multiple adjacent pixels.





https://blog.paperspace.com/a-comprehensive-exploration-of-pooling-in-neural-networks/

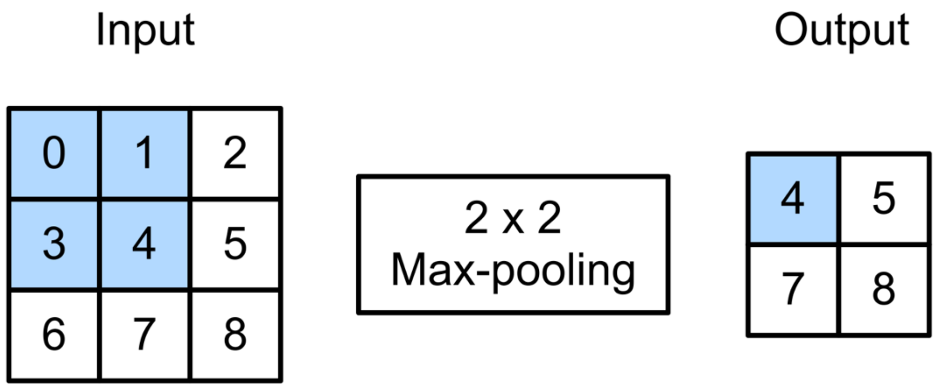
https://www.digitalocean.com/community/tutorials/pooling-in-convolutional-neural-networks

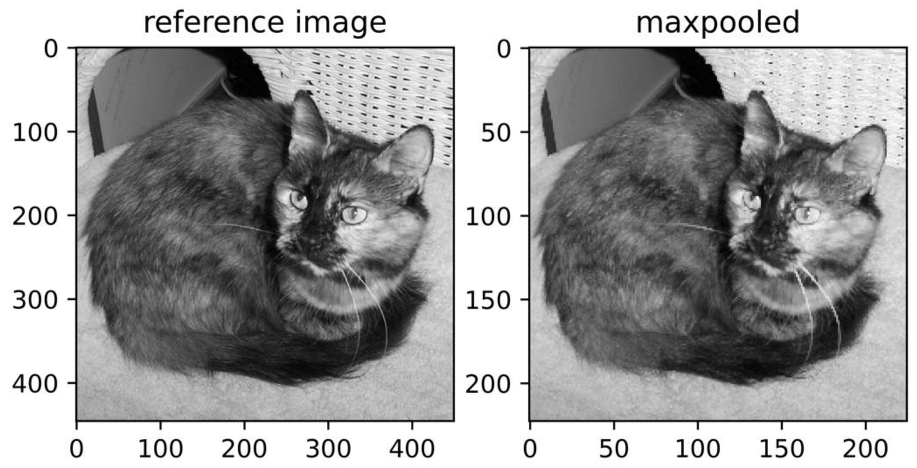
Maximum Pooling

Max-pooling was introduced in Riesenhuber and Poggio (1999) in the context of cognitive neuroscience to describe how information aggregation might be aggregated hierarchically for the purpose of object recognition; there already was an earlier version in speech recognition (Yamaguchi et al., 1990).

In almost all cases, max-pooling is preferable to average pooling.

Consider example:





https://www.digitalocean.com/community/tutorials/pooling-in-convolutional-neural-networks

Average and Maximum Pooling

Example code implements the forward propagation of the pooling layer in the pool2d function. Unlike previous corr2d function, since no kernel is needed, we compute the output as either the maximum or the average of each region in the input.

```python
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0] - p_h + 1,
X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j +
p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j +
p_w].mean()
    return Y
```

Exercises

1. Implement average pooling through a convolution.
2. Prove that max-pooling cannot be implemented through a convolution alone.
3. Max-pooling can be accomplished using ReLU operations, i.e., $ReLU(x) = \max(0, x)$.
    1. Express $\max(a, b)$ by using only ReLU operations.
    2. Use this to implement max-pooling by means of convolutions and ReLU layers.
    3. How many channels and layers do you need for a 2 x 2 convolution? How many for a 3 x 3 convolution?
4. What is the computational cost of the pooling layer? Assume that the input to the pooling layer is of size c x h x w, the pooling window has a shape of $p_h \times p_w$ with a padding of (p_h, p_w) and a stride of (s_h, s_w).
5. Why do you expect max-pooling and average pooling to work differently?
6. Do we need a separate minimum pooling layer? Can you replace it with another operation?
7. We could use the softmax operation for pooling. Why might it not be so popular?

Content

ImageNet

ImageNet

What is ImageNet?

• **Definition**: ImageNet is a large-scale visual database designed to advance research in object detection, classification, and other computer vision tasks.

• **Dataset Size**: It contains over **14 million labeled images** spanning **20,000+ categories**, with the most commonly used subset having **1,000 object categories**.

This image is CC0 1.0 public domain

This image is CC0 1.0 public domain

Key Features of ImageNet

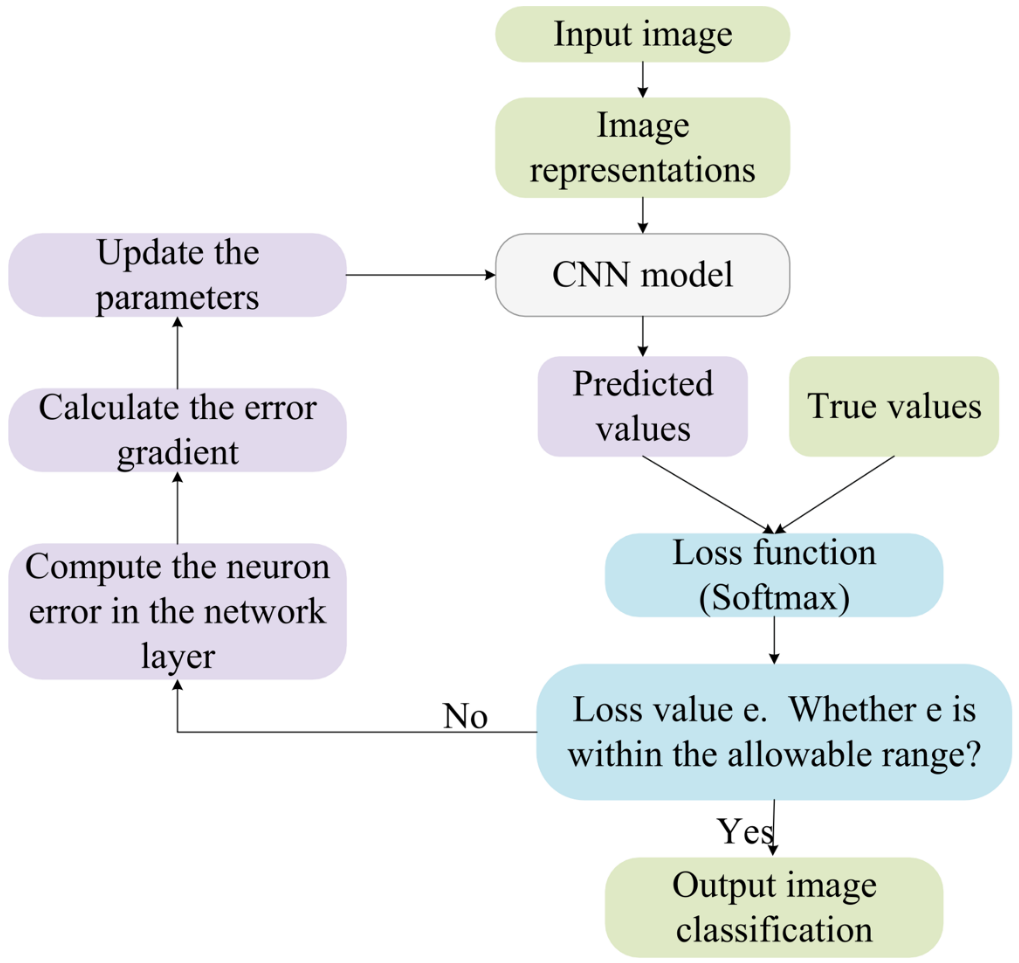
a) **Diversity of Classes**:
Includes both broad categories (e.g., "dog," "car") and fine-grained subcategories (e.g., "golden retriever," "sports car").

b) **Real-World Images**:
Images collected from the internet represent real-world complexity, including cluttered backgrounds, occlusions, and multiple objects.

c) **Hierarchical Organization**:
Based on the WordNet hierarchy, where classes are semantically related, providing meaningful relationships between categories.

CNN for Image Classification

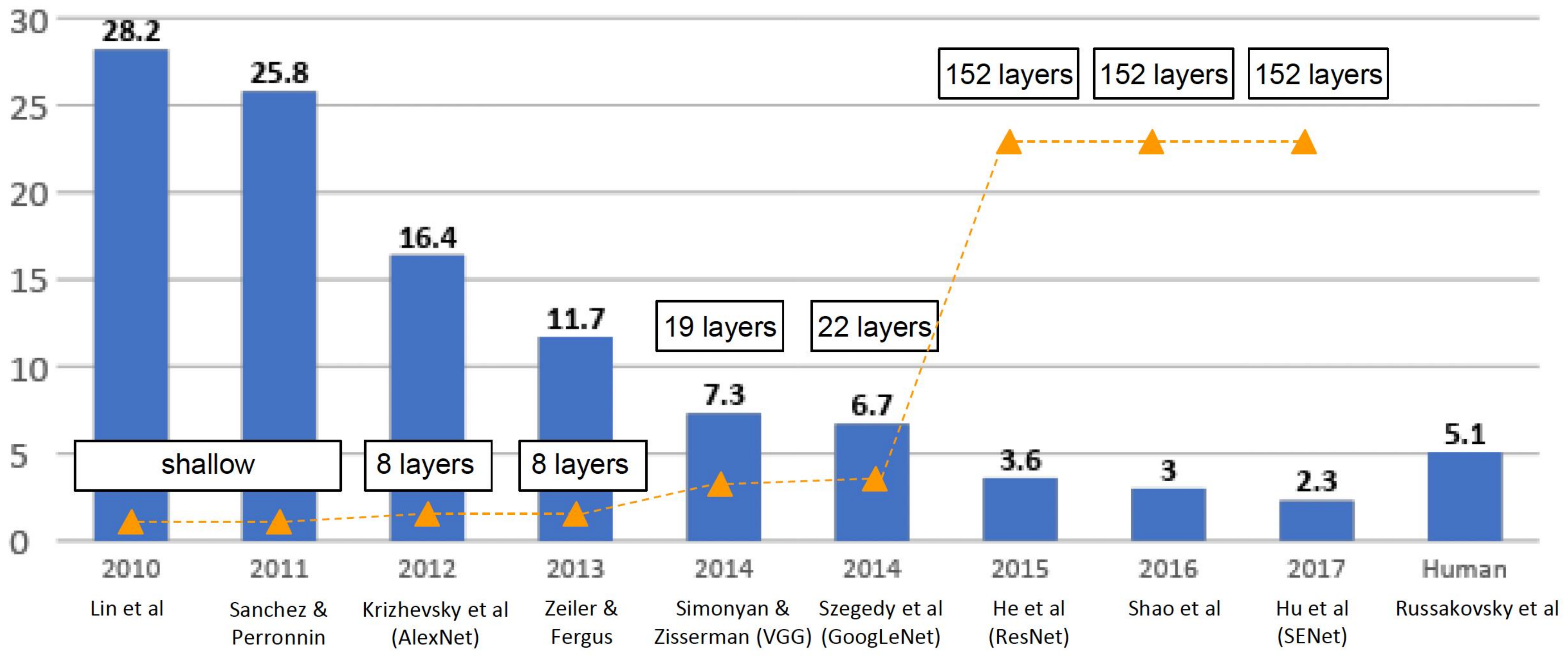
- **Fundamental Challenge**
  - Distinguishing object classes (e.g., flowers, vehicles) in images/video
  - Core steps: (1) Image preprocessing, (2) Feature extraction, (3) Classification
- **Traditional vs. CNN Approach**
  - *Traditional*:
    - Manually engineered feature extraction + classifier → Often limited in complex tasks
  - *CNN*:
    - Learns hierarchical features directly from raw inputs (via convolution kernels)
    - Scales well with large datasets → stronger generalization



Input image
Image representations
CNN model
Predicted values
True values
Loss function (Softmax)
Loss value e. Whether e is within the allowable range?
Yes
Output image classification
No
Compute the neuron error in the network layer
Calculate the error gradient
Update the parameters

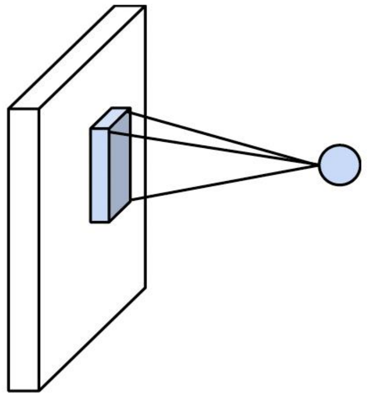
Zhao et al. *Artificial Intelligence Review* (2024)

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

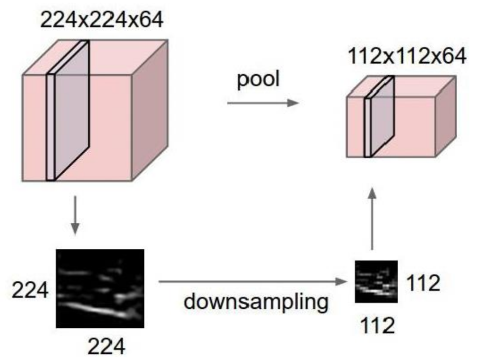


Key Components of CNNs

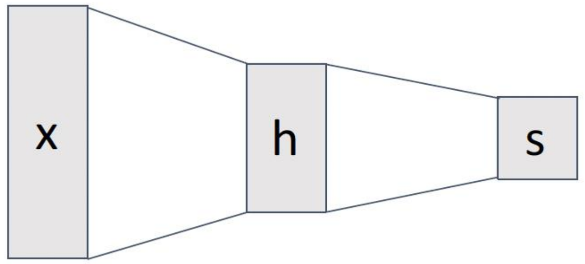
Convolution Layers



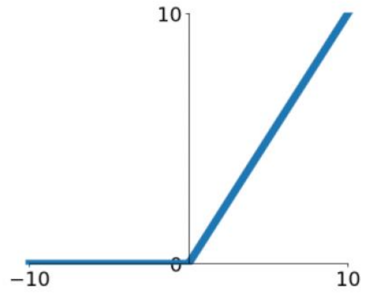
Pooling Layers



Fully-Connected Layers



Activation Function

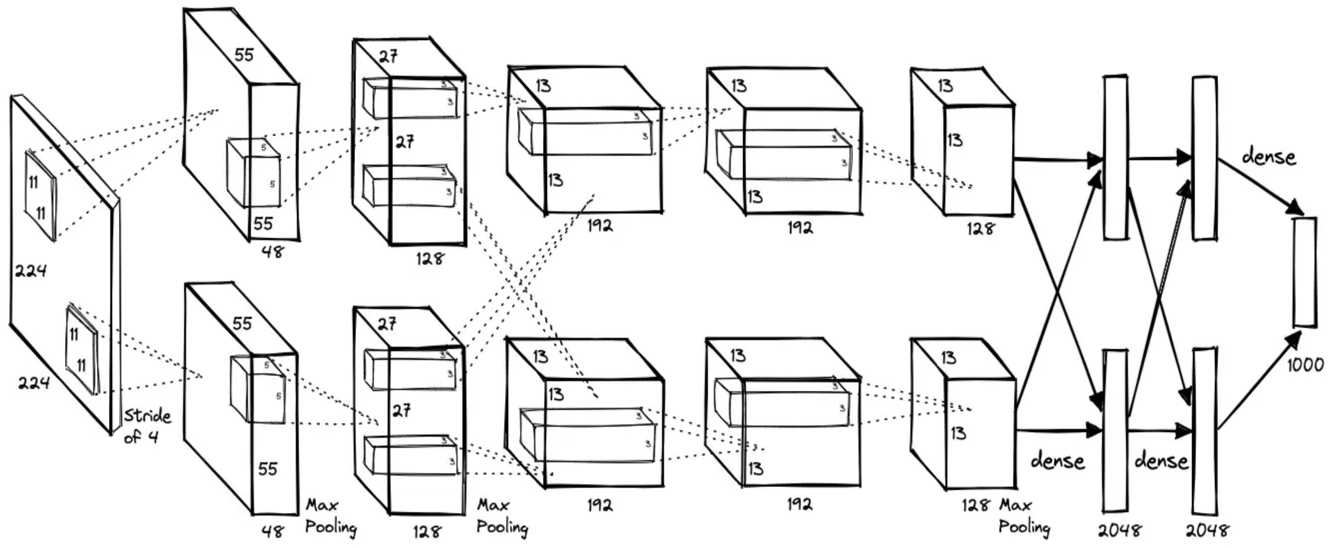


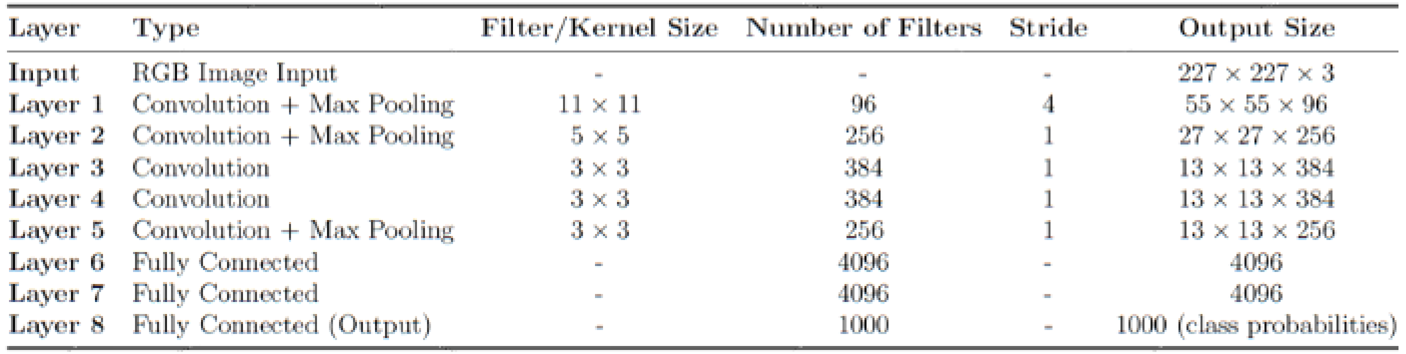
Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

AlexNet

- With high performance hardware (GPUs from Nvidia) and sufficiently rich data-set, Krizhevsky et al. proposed AlexNet (Alom et al. 2018), which consists of **five convolution layers and three fully connected layers.**
- Each convolution layer contains a convolution kernel, a bias term, a ReLU activation function, and a local response normalization (LRN) module.
- In the 2012 ILSVRC, AlexNet won the competition with a Top-5 classification error rate of **16.4%**, became the dividing line between traditional and deep learning algorithms, and was the first deep CNN model in modern times.





Layer	Type	Filter/Kernel Size	Number of Filters	Stride	Output Size
Input	RGB Image Input	-	-	-	227 × 227 × 3
Layer 1	Convolution + Max Pooling	11 × 11	96	4	55 × 55 × 96
Layer 2	Convolution + Max Pooling	5 × 5	256	1	27 × 27 × 256
Layer 3	Convolution	3 × 3	384	1	13 × 13 × 384
Layer 4	Convolution	3 × 3	384	1	13 × 13 × 384
Layer 5	Convolution + Max Pooling	3 × 3	256	1	13 × 13 × 256
Layer 6	Fully Connected	-	4096	-	4096
Layer 7	Fully Connected	-	4096	-	4096
Layer 8	Fully Connected (Output)	-	1000	-	1000 (class probabilities)

Table 1: Architecture of AlexNet (Rotated Table).

AlexNet

```
import torch
import torch.nn as nn
import torchvision.models as models

class AlexNet(nn.Module):
   def __init__(self, num_classes=1000):
   super(AlexNet, self).__init__()
  # Feature extraction layers
     self.features = nn.Sequential(
     nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),  # Conv1
     nn.ReLU(inplace=True),
     nn.MaxPool2d(kernel_size=3, stride=2),  # Pool1

    nn.Conv2d(64, 192, kernel_size=5, padding=2),  # Conv2
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=3, stride=2),  # Pool2

   nn.Conv2d(192, 384, kernel_size=3, padding=1),  # Conv3
   nn.ReLU(inplace=True),
   nn.Conv2d(384, 256, kernel_size=3, padding=1),  # Conv4
   nn.ReLU(inplace=True),
nn.Conv2d(256, 256, kernel_size=3, padding=1),  # Conv5
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=3, stride=2)  # Pool3
   )

 # Classifier layers
   self.classifier = nn.Sequential(
     nn.Dropout(),  # Dropout1
     nn.Linear(256 * 6 * 6, 4096),  # FC1
     nn.ReLU(inplace=True),
     nn.Dropout(),  # Dropout2
     nn.Linear(4096, 4096),  # FC2
     nn.ReLU(inplace=True),

     nn.Linear(4096, num_classes)  # FC3
   )

  def forward(self, x):
    # Pass through feature extraction layers
    x = self.features(x)
    x = torch.flatten(x, 1)  # Flatten the output
    # Pass through classifier layers
    x = self.classifier(x)
    return x
```

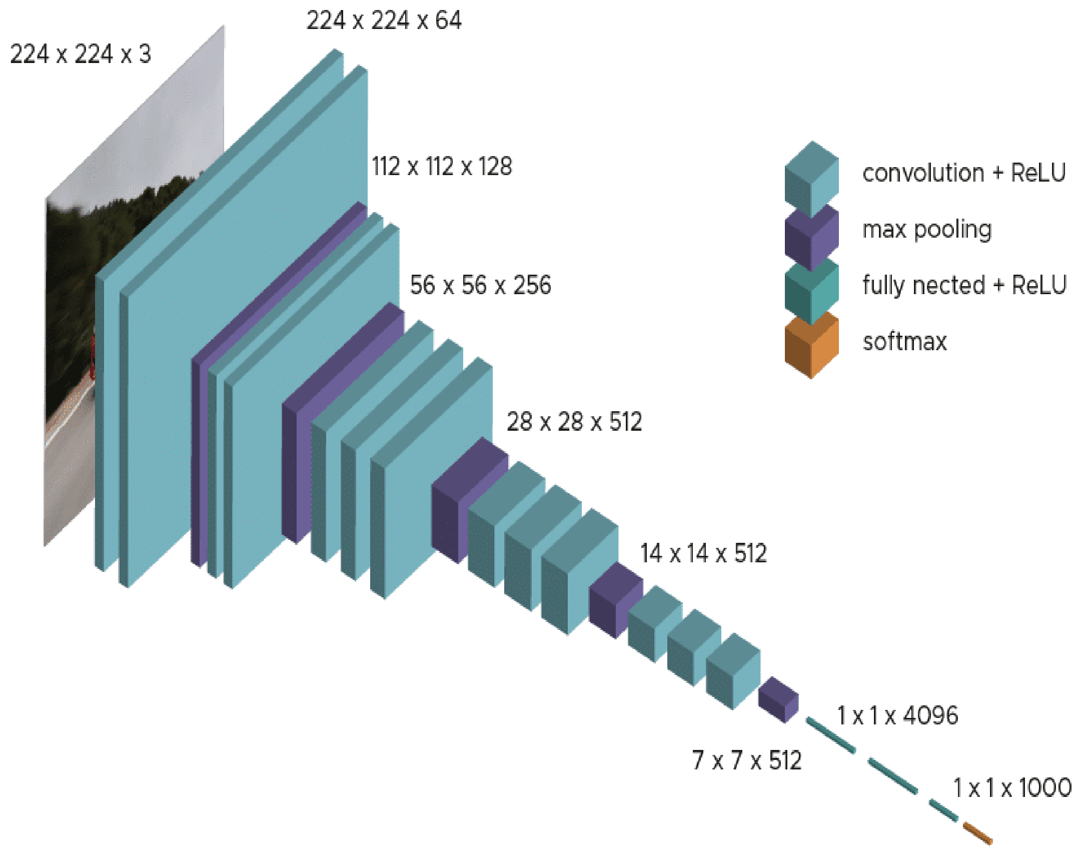
AlexNet

```
import torch
import torch.nn as nn
import torchvision.models as models

# ----- Load a pre-trained AlexNet -----
model_alexnet = models.alexnet(pretrained=True)

# ----- Forward Pass with Synthetic Data -----
# Create a single synthetic image tensor: batch size = 1, 3 color channels, 224x224
resolution
x = torch.randn(1, 3, 224, 224)
# Run the forward pass
output_alex = model_alexnet(x)

print("\nOutput shape from AlexNet:", output_alex.shape)
```

```
Output shape from AlexNet: torch.Size([1, 1000])
```

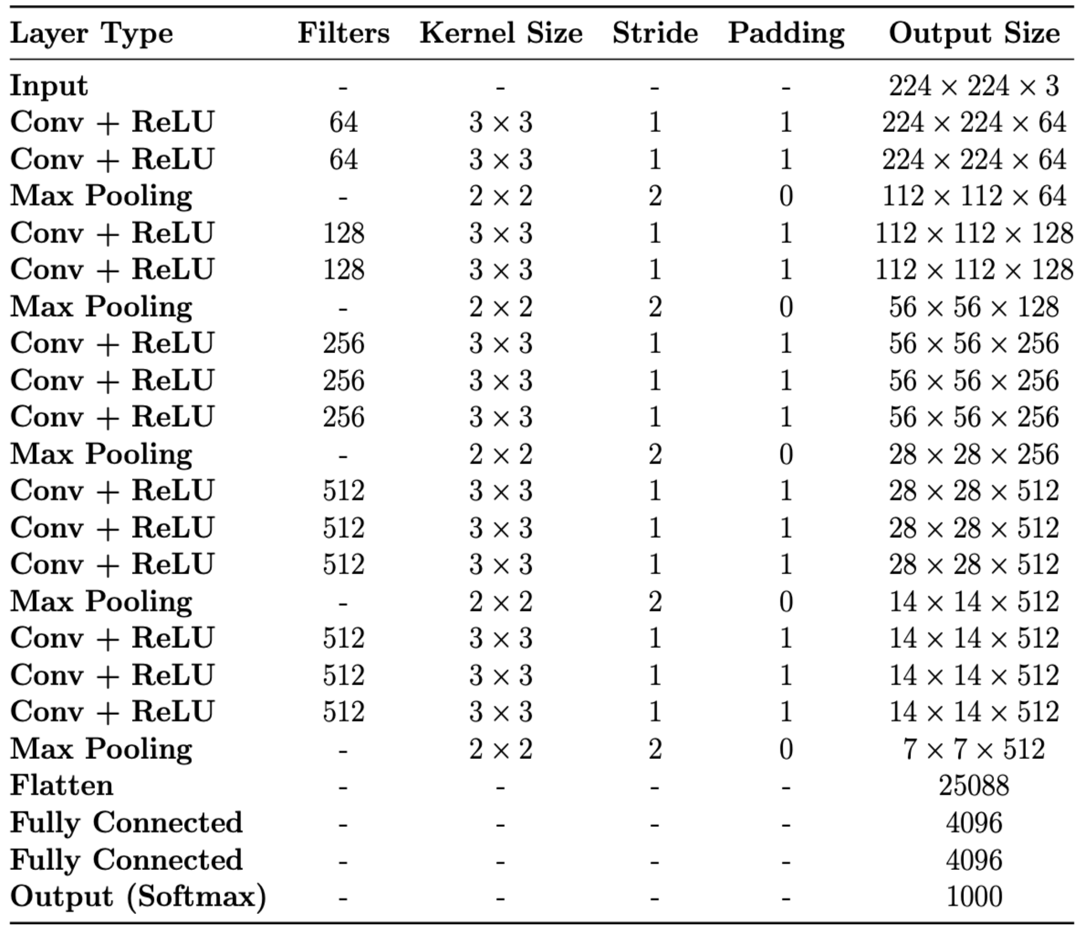
Visual Geometry Group (VGG) models

- To examine the impact of a CNN's depth on its accuracy, Karen Sengupta et al. (2019) conducted a comprehensive evaluation of the performance of network models with increasing depth, while using **smaller convolution filters (3 × 3)** instead of the previous 5 × 5 kernels and proposed a series of Visual Geometry Group (VGG) models in 2014.
- The smaller kernel size lowers the computational complexity and the number of training parameters.
- Simultaneously, VGG supports the hypothesis that performance can be enhanced by continually deepening the network topology.
- In the 2014 ILSVRC, VGG won the competition in the **Localization Task** with a Top-5 classification error rate of **7.3%**,



Sengupta et al. *Front Neurosci* (2019)

VGG Models



Layer Type	Filters	Kernel Size	Stride	Padding	Output Size
Input	-	-	-	-	224 × 224 × 3
Conv + ReLU	64	3 × 3	1	1	224 × 224 × 64
Conv + ReLU	64	3 × 3	1	1	224 × 224 × 64
Max Pooling	-	2 × 2	2	0	112 × 112 × 64
Conv + ReLU	128	3 × 3	1	1	112 × 112 × 128
Conv + ReLU	128	3 × 3	1	1	112 × 112 × 128
Max Pooling	-	2 × 2	2	0	56 × 56 × 128
Conv + ReLU	256	3 × 3	1	1	56 × 56 × 256
Conv + ReLU	256	3 × 3	1	1	56 × 56 × 256
Conv + ReLU	256	3 × 3	1	1	56 × 56 × 256
Max Pooling	-	2 × 2	2	0	28 × 28 × 256
Conv + ReLU	512	3 × 3	1	1	28 × 28 × 512
Conv + ReLU	512	3 × 3	1	1	28 × 28 × 512
Conv + ReLU	512	3 × 3	1	1	28 × 28 × 512
Max Pooling	-	2 × 2	2	0	14 × 14 × 512
Conv + ReLU	512	3 × 3	1	1	14 × 14 × 512
Conv + ReLU	512	3 × 3	1	1	14 × 14 × 512
Conv + ReLU	512	3 × 3	1	1	14 × 14 × 512
Max Pooling	-	2 × 2	2	0	7 × 7 × 512
Flatten	-	-	-	-	25088
Fully Connected	-	-	-	-	4096
Fully Connected	-	-	-	-	4096
Output (Softmax)	-	-	-	-	1000

Table 1: VGG-16 Architecture: Layers, filters, and output sizes.

a) **Increased Depth**:
   Depth allows VGG to learn hierarchical features, improving accuracy.

b) **Simple Design**:
   Stacks of identical convolutional layers make it easy to scale the architecture.

c) **Transfer Learning**:
   VGG models pretrained on ImageNet are widely used for transfer learning in other tasks.

d) **Small Filters**:
   Using 3×3 filters results in fewer parameters compared to larger filters, while maintaining the receptive field size.

e) **VGG-16**:

16 layers: 13 convolutional layers and 3 fully connected layers.
Parameters: ~138 million.

f) **VGG-19**:

19 layers: 16 convolutional layers and 3 fully connected layers.
Parameters: ~143 million.

VGG models

```
import torch
import torch.nn as nn
import torchvision.models as models

class VGG16(nn.Module):
   def __init__(self, num_classes=1000):
   super(VGG16, self).__init__()
   # Feature extraction layers
   self.features = nn.Sequential(
   # Block 1
   nn.Conv2d(3, 64, kernel_size=3, padding=1),  # Conv1_1
   nn.ReLU(inplace=True),
   nn.Conv2d(64, 64, kernel_size=3, padding=1),  # Conv1_2
   nn.ReLU(inplace=True),
   nn.MaxPool2d(kernel_size=2, stride=2),  # Pool1
   # Block 2
   nn.Conv2d(64, 128, kernel_size=3, padding=1), # Conv2_1
   nn.ReLU(inplace=True),
   nn.Conv2d(128, 128, kernel_size=3, padding=1), # Conv2_2
   nn.ReLU(inplace=True),
   nn.MaxPool2d(kernel_size=2, stride=2),  # Pool2
   # Block 3
   nn.Conv2d(128, 256, kernel_size=3, padding=1),  # Conv3_1
   nn.ReLU(inplace=True),
   nn.Conv2d(256, 256, kernel_size=3, padding=1),  # Conv3_2
   nn.ReLU(inplace=True),
   nn.Conv2d(256, 256, kernel_size=3, padding=1),  # Conv3_3
   nn.ReLU(inplace=True),
   nn.MaxPool2d(kernel_size=2, stride=2),  # Pool3
```

```
# Block 4
n.Conv2d(256, 512, kernel_size=3, padding=1),  # Conv4_1
n.ReLU(inplace=True),
n.Conv2d(512, 512, kernel_size=3, padding=1),  # Conv4_2
n.ReLU(inplace=True),
n.Conv2d(512, 512, kernel_size=3, padding=1),  # Conv4_3
n.ReLU(inplace=True),
n.MaxPool2d(kernel_size=2, stride=2),  # Pool4
 Block 5
n.Conv2d(512, 512, kernel_size=3, padding=1),  # Conv5_1
n.ReLU(inplace=True),
n.Conv2d(512, 512, kernel_size=3, padding=1),  # Conv5_2
n.ReLU(inplace=True),
n.Conv2d(512, 512, kernel_size=3, padding=1),  # Conv5_3
n.ReLU(inplace=True),
n.MaxPool2d(kernel_size=2, stride=2)  # Pool5

 Classification layers
       self.classifier = nn.Sequential(
           nn.Linear(512 * 7 * 7, 4096),  # FC1
           nn.ReLU(inplace=True),
           nn.Dropout(),
           nn.Linear(4096, 4096),  # FC2
           nn.ReLU(inplace=True),
           nn.Dropout(),
           nn.Linear(4096, num_classes)  # FC3
       )
```

VGG models

```
def forward(self, x):
        # Pass through feature extraction layers
        x = self.features(x)
        x = torch.flatten(x, 1)  # Flatten the output
        # Pass through classifier layers
        x = self.classifier(x)
        return x

# Example usage
if __name__ == "__main__":
    # Initialize the model
    model = VGG16(num_classes=1000)
    print(model)

    # Test with a random input
    input_tensor = torch.randn(1, 3, 224, 224)  # Batch size = 1, 3 channels, 224x224 image
    output = model(input_tensor)
    print("Output shape:", output.shape)  # Should be [1, 1000] for 1000 classes
```

GoogLeNet

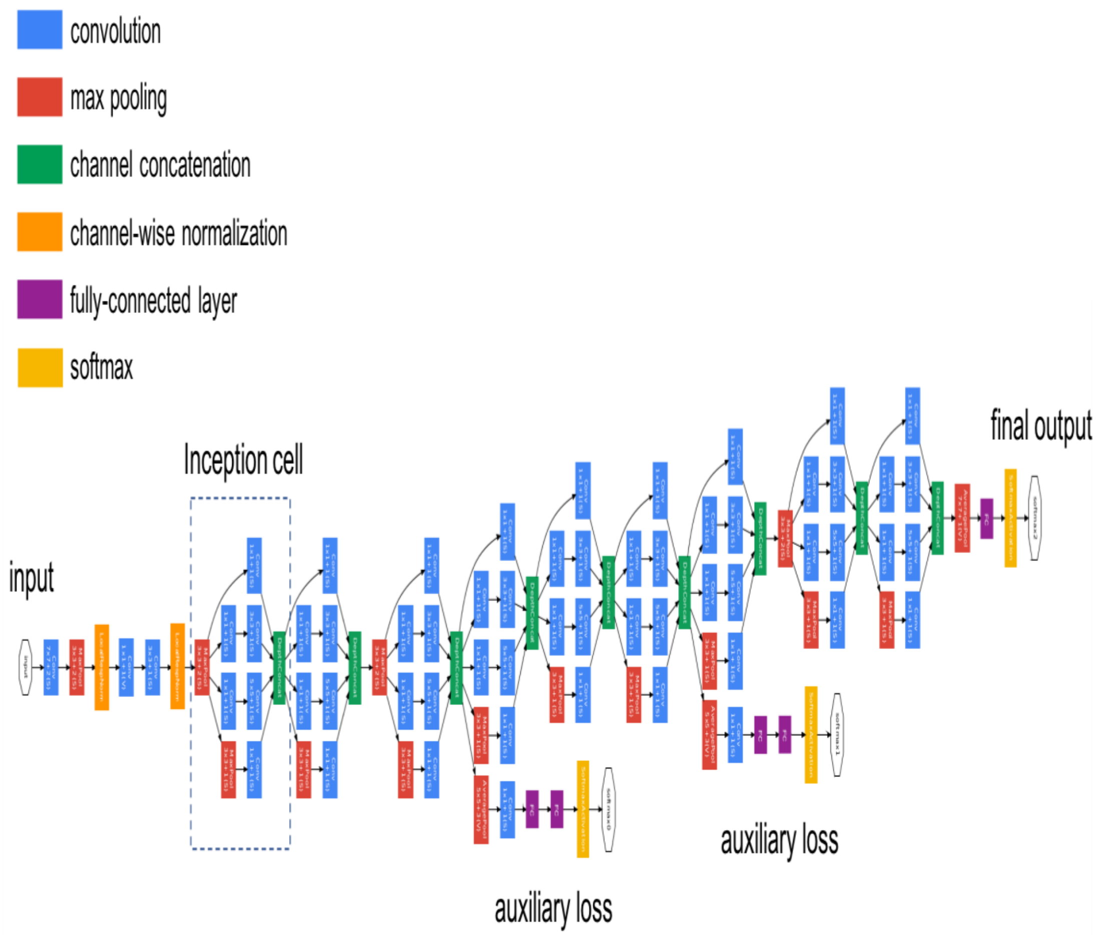
- GoogleNet, also known as **Inception-v1**, is a deep CNN introduced by Szegedy et al. in 2014.
- It won the **ILSVRC** 2014 the **Classification Task** with a top-5 error rate of **6.67%**, outperforming other models.
- **Main Innovations**:

a) **Inception Module** enables the network to capture features at multiple scales while reducing computational cost.

b) **Dimension Reduction.** Uses 1×1 convolutions for reducing dimensionality before applying larger filters, significantly reducing parameters.

c) **Auxiliary Classifiers**: Two intermediate softmax classifiers are added to help with gradient flow and prevent vanishing gradients.

- **Motivation**: Despite having 22 layers, GoogleNet has only **~5M parameters**, significantly fewer than AlexNet (~60M) and VGG-16 (~138M).This is achieved using 1×11×1 convolutions for dimensionality reduction.



Szegedy et al. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015)

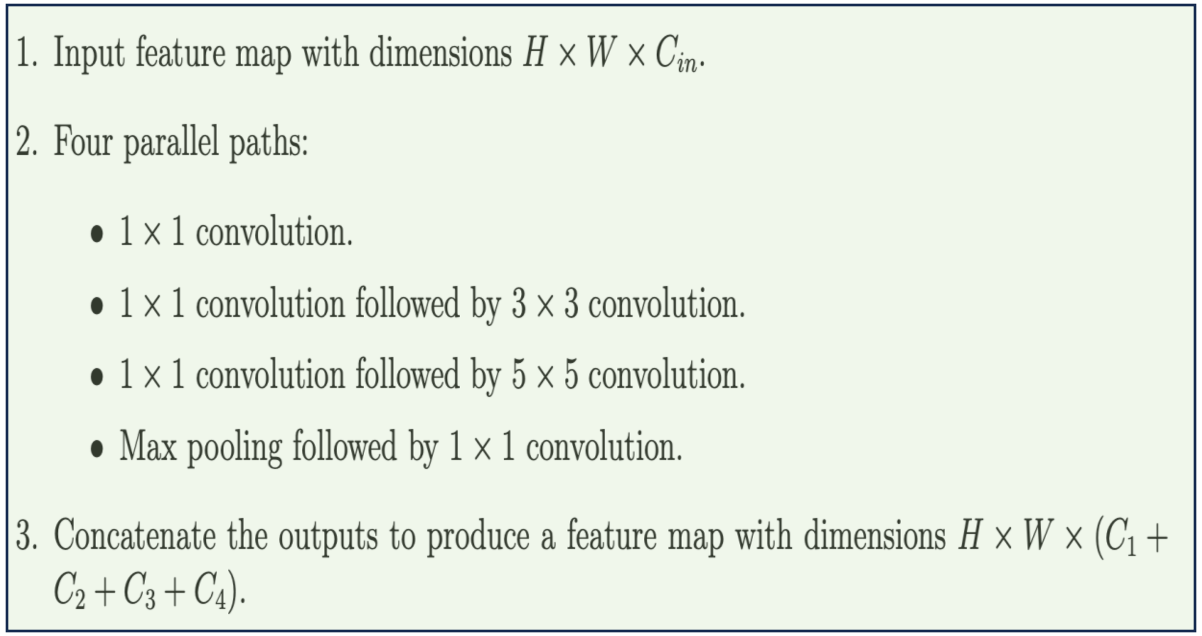
GoogleNet Architecture

Layer Type	Filters/Units	Kernel Size	Stride	Padding	Output Size
Input	-	-	-	-	$224 \times 224 \times 3$
Conv + ReLU	64	7×7	2	3	$112 \times 112 \times 64$
Max Pooling	-	3×3	2	0	$56 \times 56 \times 64$
Conv + ReLU	64	1×1	1	0	$56 \times 56 \times 64$
Conv + ReLU	192	3×3	1	1	$56 \times 56 \times 192$
Max Pooling	-	3×3	2	0	$28 \times 28 \times 192$
Inception Module 1	-	Multi-scale	-	-	$28 \times 28 \times 256$
Inception Module 2	-	Multi-scale	-	-	$28 \times 28 \times 480$
Max Pooling	-	3×3	2	0	$14 \times 14 \times 480$
Inception Module 3	-	Multi-scale	-	-	$14 \times 14 \times 512$
Inception Module 4	-	Multi-scale	-	-	$14 \times 14 \times 512$
Inception Module 5	-	Multi-scale	-	-	$14 \times 14 \times 528$
Auxiliary Classifier 1	1000	-	-	-	1000
Inception Module 6	-	Multi-scale	-	-	$14 \times 14 \times 832$
Auxiliary Classifier 2	1000	-	-	-	1000
Inception Module 7	-	Multi-scale	-	-	$7 \times 7 \times 1024$
Global Average Pooling	-	7×7	-	-	$1 \times 1 \times 1024$
Fully Connected	1000	-	-	-	1000

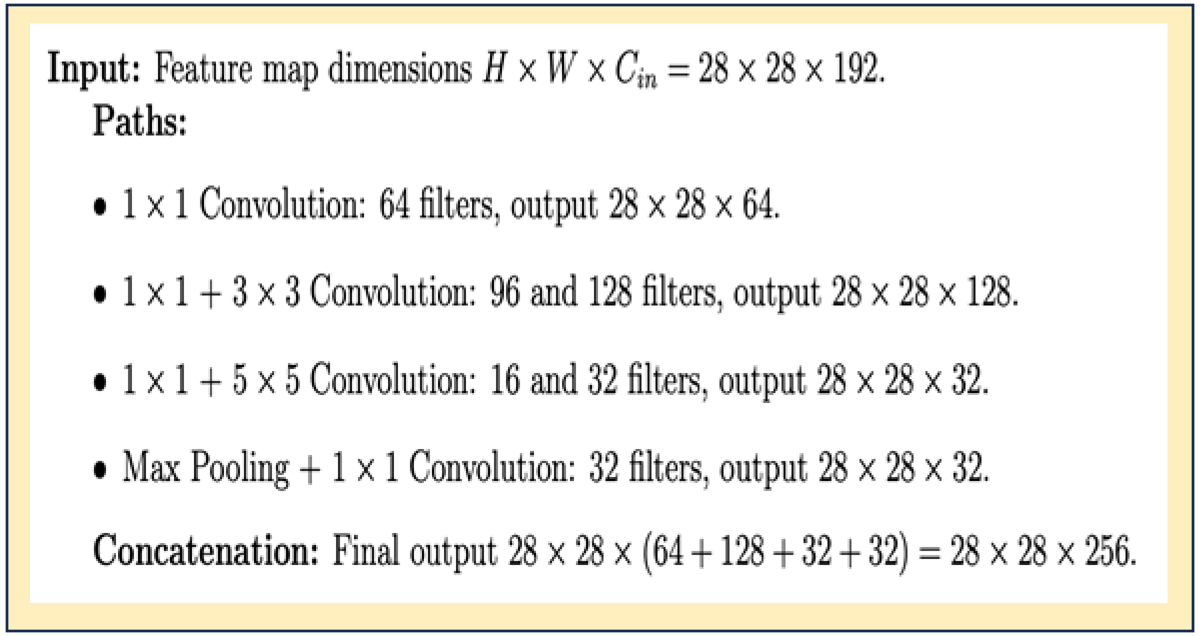
Table 1: GoogleNet Architecture: Layers, filters, and output sizes.

•**Input Layer**: 224×224×224×3 RGB image.
•**Convolutional Layers**: Apply 7×7, 1×1, or 3×3 filters to extract features.
•**Inception Modules**: Multi-scale processing with 1×1, 3×3, 5×5, and pooling operations.
•**Auxiliary Classifiers**: Intermediate softmax layers for training regularization.
•**Global Average Pooling**: Replaces fully connected layers with spatial pooling across feature maps.
Output Sizes:
•The output size at each stage is shown, demonstrating how spatial dimensions decrease progressively.

Inception Module



1. Input feature map with dimensions $H \times W \times C_{in}$.
2. Four parallel paths:
   - 1×1 convolution.
   - 1×1 convolution followed by 3×3 convolution.
   - 1×1 convolution followed by 5×5 convolution.
   - Max pooling followed by 1×1 convolution.
3. Concatenate the outputs to produce a feature map with dimensions $H \times W \times (C_1 + C_2 + C_3 + C_4)$.



Input: Feature map dimensions $H \times W \times C_{in} = 28 \times 28 \times 192$.

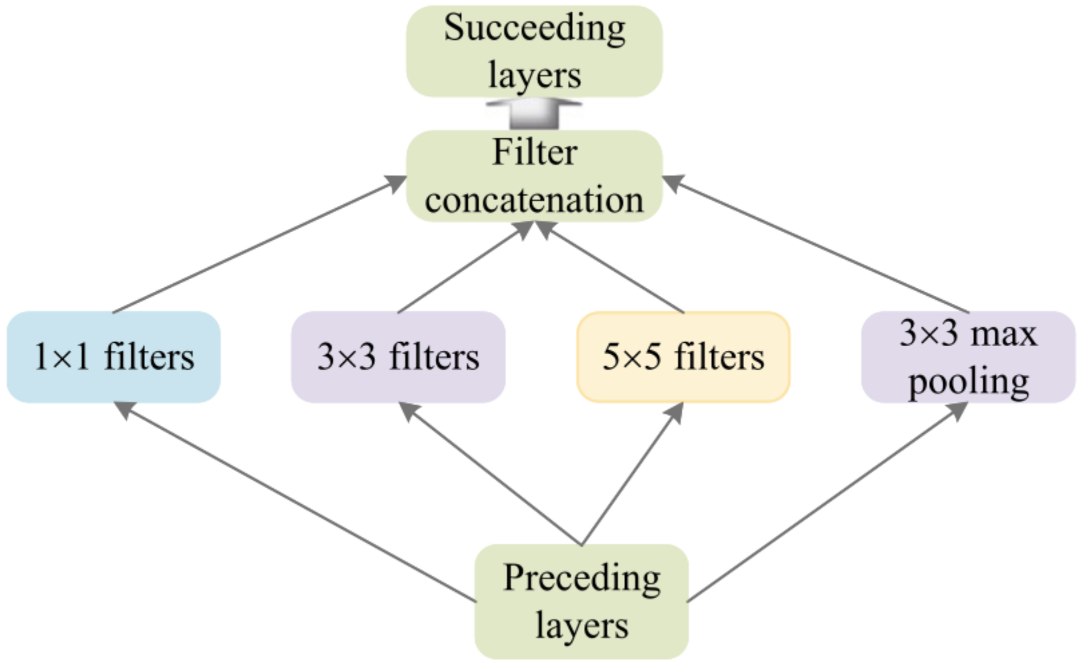
Paths:

- 1×1 Convolution: 64 filters, output $28 \times 28 \times 64$.
- $1 \times 1 + 3 \times 3$ Convolution: 96 and 128 filters, output $28 \times 28 \times 128$.
- $1 \times 1 + 5 \times 5$ Convolution: 16 and 32 filters, output $28 \times 28 \times 32$.
- Max Pooling $+ 1 \times 1$ Convolution: 32 filters, output $28 \times 28 \times 32$.

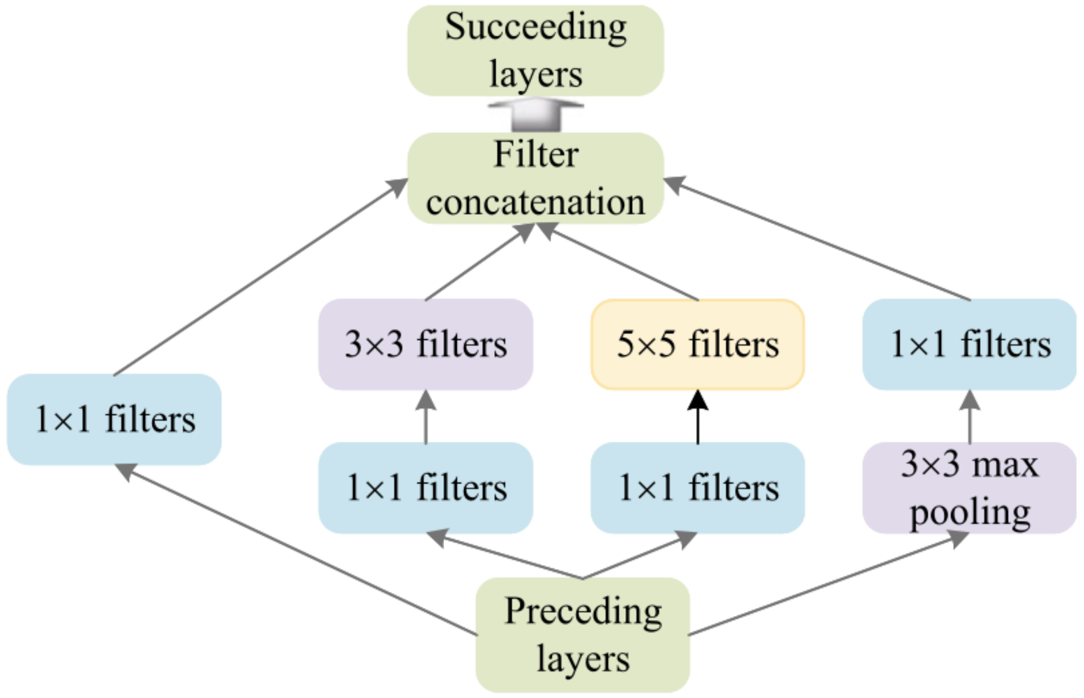
Concatenation: Final output $28 \times 28 \times (64 + 128 + 32 + 32) = 28 \times 28 \times 256$.

- **Multi-Scale Feature Extraction:** Processes feature maps at multiple scales for rich. representations.
- **Dimensionality Reduction:** 1×1 convolutions reduce computational costs while preserving important information.
- **Efficiency:** Deep networks can process large input data with fewer parameters compared to traditional architectures.
- **Improved Generalization:** Captures features across different abstraction levels.

Inception Cell



(a) Architecture of inception



(b) Architecture of inception V1

Example architecture of inception
Zhao et al. *Artificial Intelligence Review* (2024)

GoogLeNet

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Inception(nn.Module):
    def __init__(self, in_channels, ch1x1, ch3x3_reduce, ch3x3, ch5x5_reduce, ch5x5, pool_proj):
        """
        Inception Module
        Args:
            in_channels: Input channels
            ch1x1: Number of filters for 1x1 convolution
            ch3x3_reduce: Number of filters for 1x1 convolution before 3x3 convolution
            ch3x3: Number of filters for 3x3 convolution
            ch5x5_reduce: Number of filters for 1x1 convolution before 5x5 convolution
            ch5x5: Number of filters for 5x5 convolution
            pool_proj: Number of filters for the projection from pooling
        """
        super(Inception, self).__init__()
        # 1x1 Convolution
        self.branch1 = nn.Sequential(
            nn.Conv2d(in_channels, ch1x1, kernel_size=1, bias=False),
            nn.BatchNorm2d(ch1x1),
            nn.ReLU(inplace=True) )
```

```
# 1x1 Convolution -> 3x3 Convolution
        self.branch2 = nn.Sequential(
            nn.Conv2d(in_channels, ch3x3_reduce, kernel_size=1, bias=False),
            nn.BatchNorm2d(ch3x3_reduce),
            nn.ReLU(inplace=True),
            nn.Conv2d(ch3x3_reduce, ch3x3, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(ch3x3),
            nn.ReLU(inplace=True)
        )
        # 1x1 Convolution -> 5x5 Convolution
        self.branch3 = nn.Sequential(
            nn.Conv2d(in_channels, ch5x5_reduce, kernel_size=1, bias=False),
            nn.BatchNorm2d(ch5x5_reduce),
            nn.ReLU(inplace=True),
            nn.Conv2d(ch5x5_reduce, ch5x5, kernel_size=5, padding=2, bias=False),
            nn.BatchNorm2d(ch5x5),
            nn.ReLU(inplace=True)
        )
        # 3x3 Pooling -> 1x1 Convolution
        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
            nn.Conv2d(in_channels, pool_proj, kernel_size=1, bias=False),
            nn.BatchNorm2d(pool_proj),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)
        return torch.cat([branch1, branch2, branch3, branch4], 1)
```

GoogLeNet

```
class GoogleNet(nn.Module):
  def __init__(self, num_classes=1000):
 super(GoogleNet, self).__init__()
    # Initial layers
    self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
    self.bn1 = nn.BatchNorm2d(64)
    self.relu = nn.ReLU(inplace=True)
    self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    # Convolutions and pooling layers
    self.conv2 = nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1, bias=False)
    self.bn2 = nn.BatchNorm2d(192)
    self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    # Inception modules
    self.inception3a = Inception(192, 64, 96, 128, 16, 32, 32)
    self.inception3b = Inception(256, 128, 128, 192, 32, 96, 64)
    self.maxpool3 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

    self.inception4a = Inception(480, 192, 96, 208, 16, 48, 64)
    self.inception4b = Inception(512, 160, 112, 224, 24, 64, 64)
    self.inception4c = Inception(512, 128, 128, 256, 24, 64, 64)
    self.inception4d = Inception(512, 112, 144, 288, 32, 64, 64)
    self.inception4e = Inception(528, 256, 160, 320, 32, 128, 128)
    self.maxpool4 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

    self.inception5a = Inception(832, 256, 160, 320, 32, 128, 128)
    self.inception5b = Inception(832, 384, 192, 384, 48, 128, 128)
```

```
# Global Average Pooling and Fully Connected Layer
    self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
    self.fc = nn.Linear(1024, num_classes)

  def forward(self, x):
    x = self.relu(self.bn1(self.conv1(x)))
    x = self.maxpool1(x)

    x = self.bn2(self.conv2(x))
    x = self.maxpool2(x)

    x = self.inception3a(x)
    x = self.inception3b(x)
    x = self.maxpool3(x)

    x = self.inception4a(x)
    x = self.inception4b(x)
    x = self.inception4c(x)
    x = self.inception4d(x)
    x = self.inception4e(x)
    x = self.maxpool4(x)

    x = self.inception5a(x)
    x = self.inception5b(x)

    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)
    return x
```

GoogLeNet

```
import torch
import torch.nn as nn
import torchvision.models as models

# Example usage
if __name__ == "__main__":
   # Initialize the model
   model = GoogleNet(num_classes=1000)
   print(model)

   # Test with a random input
   input_tensor = torch.randn(1, 3, 224, 224)  # Batch size = 1, 3 channels, 224x224 image
   output = model(input_tensor)
   print("Output shape:", output.shape)  # Should be [1, 1000] for 1000 classes
```

Residual network (ResNet)

- **Degradation Problem:** Deeper networks (e.g., >20 layers) suffered from degradation of accuracy, not just overfitting, but actual performance decline.
- **Key Idea:** Instead of learning the direct mapping (H(x)), ResNet learns the **residual mapping** $(F(x)=H(x)-x)$. This simplifies optimization and allows gradients to flow through skip connections, improving convergence.
- **Impact:**
  - **Ease of Optimization:** Learning residuals is simpler than learning direct mappings.
  - **Deeper Architectures:** ResNet-152 outperforms shallower networks while maintaining high accuracy.
  - **State-of-the-art Results:** Top-5 error dropped to **~3.6%** on ImageNet (ILSVRC).
- **Connection to Highway Networks (Srivastava et al., 2015):** ResNet can be seen as a special, simplified case of highway layers where gates are mostly open.
- Residual connections enable building much deeper and more powerful networks by addressing gradient vanishing and "degradation" issues.

Residual Block

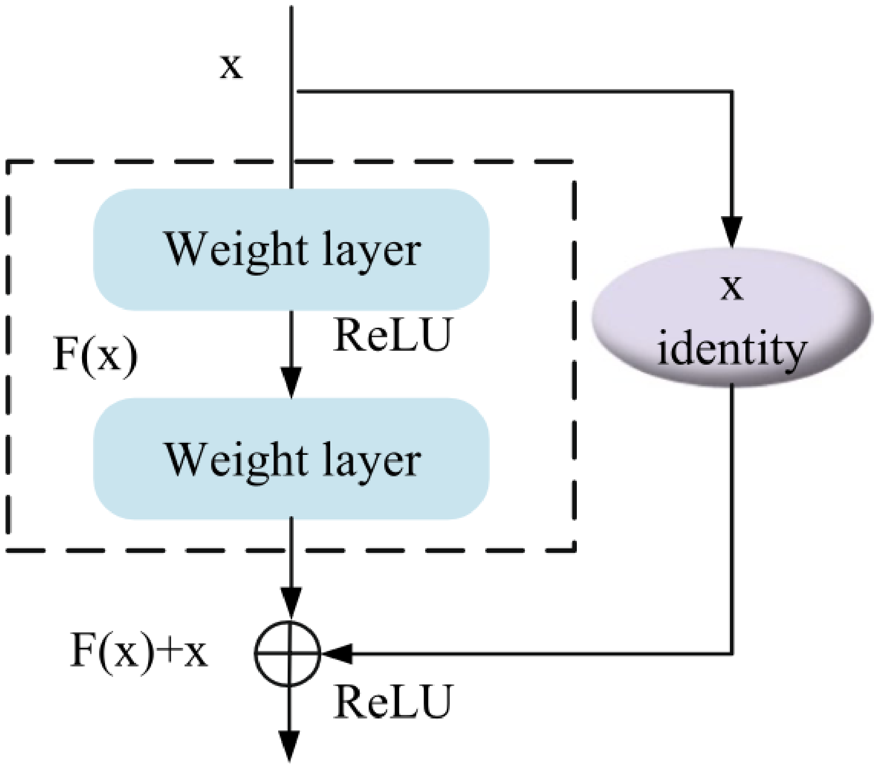


Illustration of a residual block
Zhao et al. *Artificial Intelligence Review* (2024)

Building Block:

a) **Input**: x (feature map from the previous layer).

b) **Path 1 (Residual Function)**:
   i. 3×3 convolution -> Batch Normalization -> ReLU.
   ii. 3×3 convolution -> Batch Normalization.

c) **Path 2 (Skip Connection)**:
   i. Identity mapping: Directly passes the input x.

d) **Addition**:
   i. Output: F(x)+x (summation of the two paths).

e) **Activation**:
   i. Apply ReLU to the combined output.

f) **Output:**
Final feature map retains the same dimensions as the input.

Residual Block

```
import torch
import torch.nn as nn
import torchvision.models as models

# Define a Residual Block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
# Skip connection
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
```

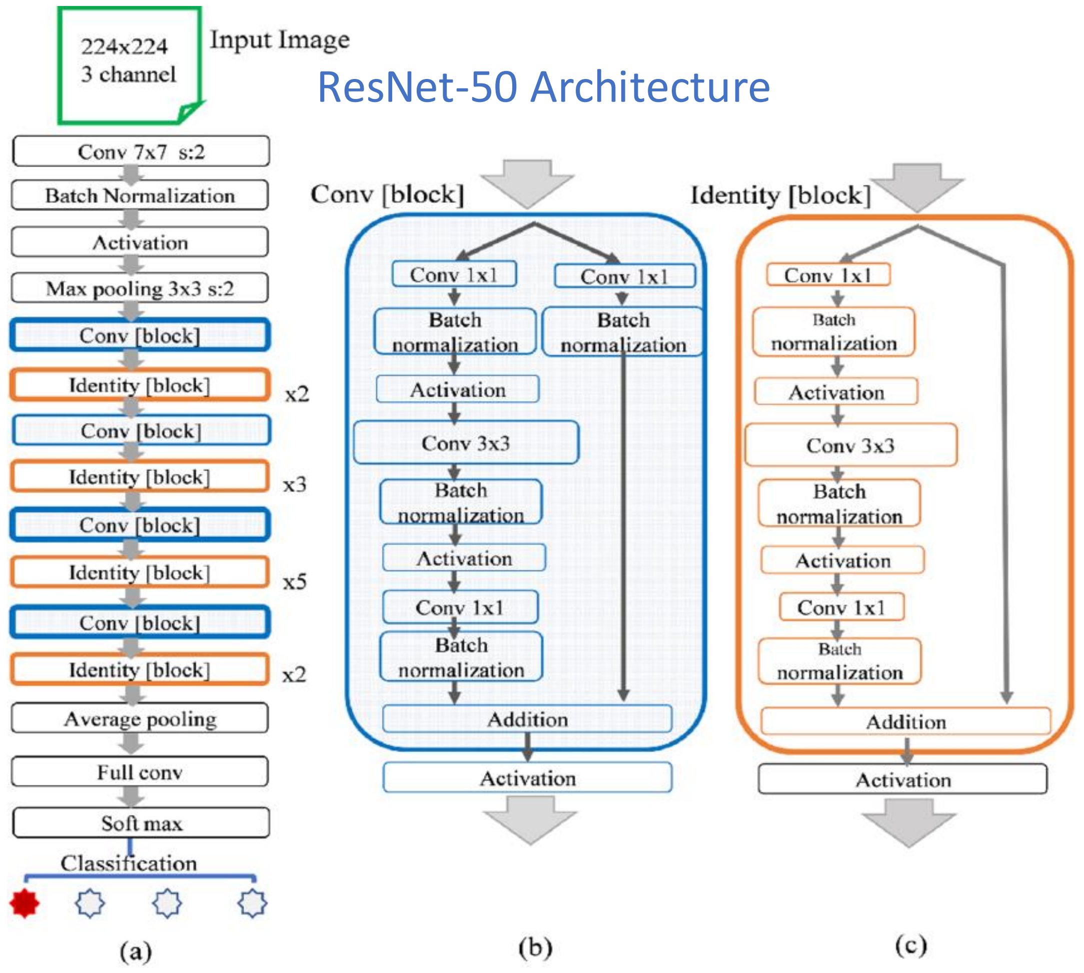
```
def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)
        out = nn.ReLU()(out)
        out = self.conv2(out)
       out = self.bn2(out)
       out += self.shortcut(x)
        out = nn.ReLU()(out)
        return out
```

ResNet-50

Stage	Type	Number of Blocks	Filters	Output Size
Input	RGB Image	-	-	224 × 224 × 3
Conv1	7 × 7 Conv, Stride 2	-	64	112 × 112 × 64
Pooling	3 × 3 Max Pool, Stride 2	-	-	56 × 56 × 64
Conv2_x	Residual Block	3	64, 64, 256	56 × 56 × 256
Conv3_x	Residual Block	4	128, 128, 512	28 × 28 × 512
Conv4_x	Residual Block	6	256, 256, 1024	14 × 14 × 1024
Conv5_x	Residual Block	3	512, 512, 2048	7 × 7 × 2048
Pooling	Global Avg Pooling	-	-	1 × 1 × 2048
Output	Fully Connected (Dense)	-	1000 (classes)	1000

Table 1: ResNet-50 Architecture: Stages, block types, filters, and output sizes.

ResNet-50 Architecture

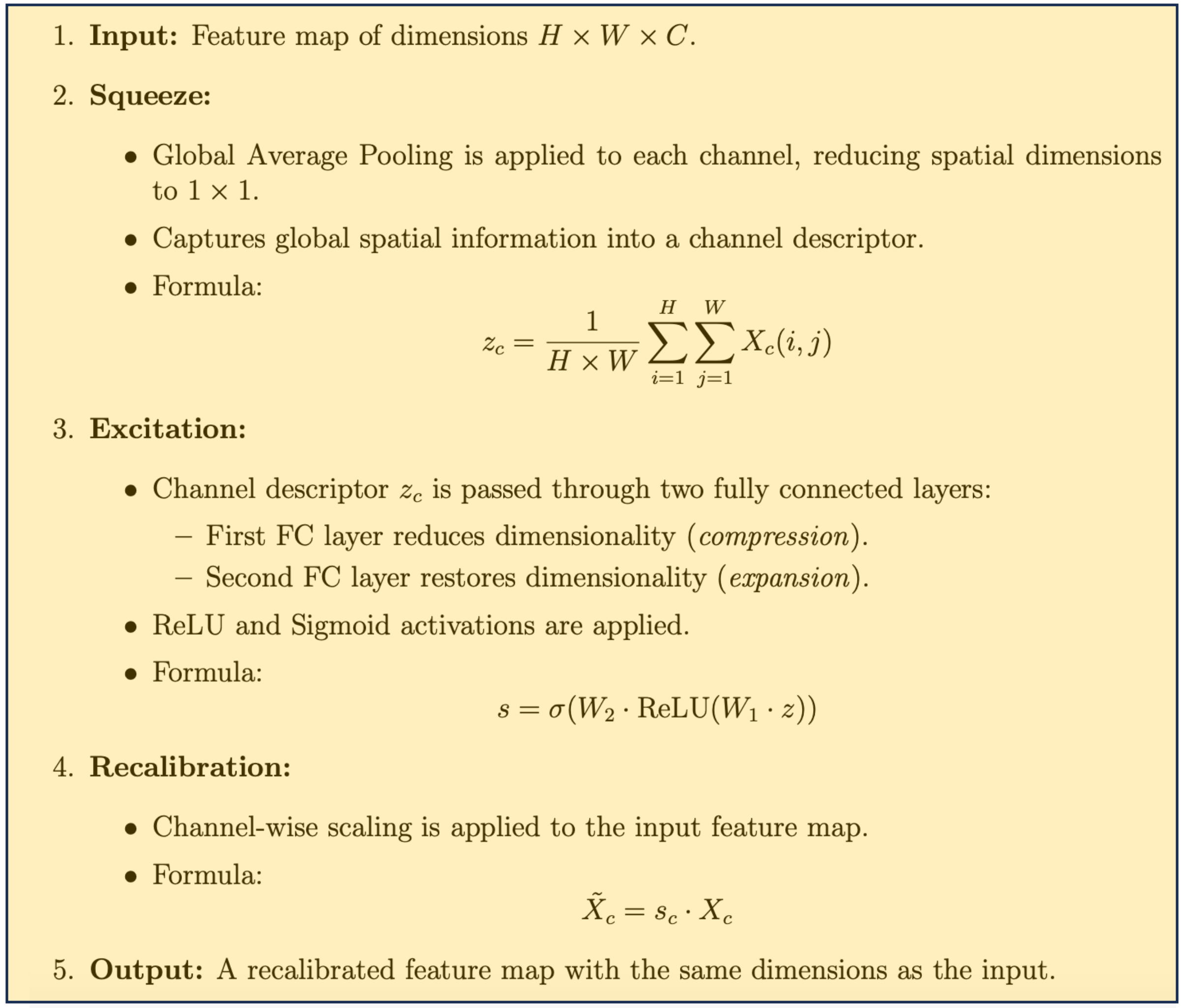


ResNet-50

```
# Define ResNet-50
class ResNet50(nn.Module):
  def __init__(self, num_classes=1000):
    super(ResNet50, self).__init__()
    self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
    self.bn1 = nn.BatchNorm2d(64)
    self.relu = nn.ReLU()
    self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

    # Define ResNet Blocks
    self.layer1 = self._make_layer(64, 256, 3)
    self.layer2 = self._make_layer(256, 512, 4, stride=2)
    self.layer3 = self._make_layer(512, 1024, 6, stride=2)
    self.layer4 = self._make_layer(1024, 2048, 3, stride=2)

    self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
    self.fc = nn.Linear(2048, num_classes)

  def _make_layer(self, in_channels, out_channels, blocks, stride=1):
    layers = [ResidualBlock(in_channels, out_channels, stride)]
    for _ in range(1, blocks):
      layers.append(ResidualBlock(out_channels, out_channels))
    return nn.Sequential(*layers)

def forward(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)
    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)
    return x
# Instantiate and print the model
 model = ResNet50()
 print(model)
```

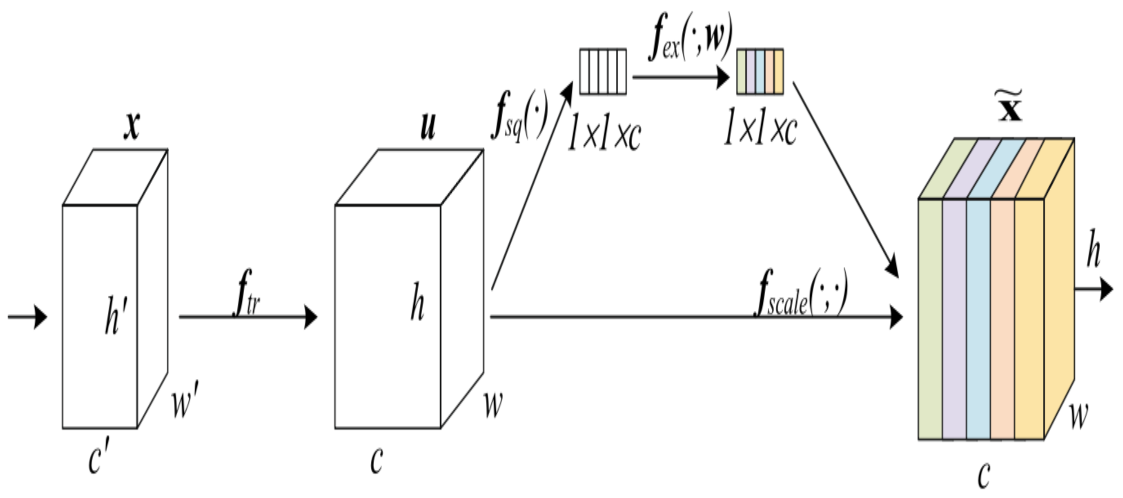
Squeeze-and-Excitation (SE) Networks

- **Definition**: Squeeze-and-Excitation (SE) Networks are CNN architectural units introduced by Hu et al. in the paper "Squeeze-and-Excitation Networks" (CVPR 2018).
- **Objective**: To improve channel-wise feature recalibration by explicitly modeling interdependencies between feature channels.
- **Motivation**: Standard convolutional operations treat all channels equally, potentially ignoring the inter-channel dependencies.
- SE blocks enhance channel sensitivity, allowing the network to emphasize important features and suppress irrelevant ones.

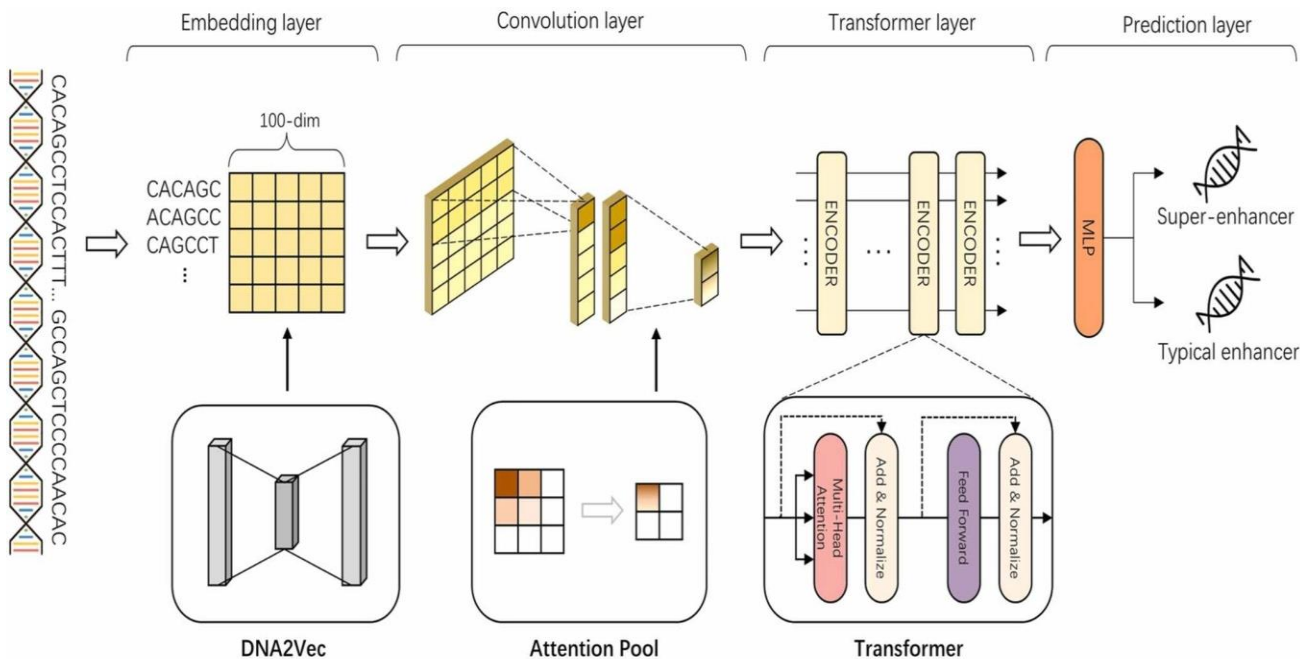


1. **Input:** Feature map of dimensions $H \times W \times C$.
2. **Squeeze:**
   - Global Average Pooling is applied to each channel, reducing spatial dimensions to 1×1.
   - Captures global spatial information into a channel descriptor.
   - Formula:
   $$z_c = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} X_c(i, j)$$
3. **Excitation:**
   - Channel descriptor z_c is passed through two fully connected layers:
     - First FC layer reduces dimensionality (*compression*).
     - Second FC layer restores dimensionality (*expansion*).
   - ReLU and Sigmoid activations are applied.
   - Formula:
   $$s = \sigma(W_2 \cdot \text{ReLU}(W_1 \cdot z))$$
4. **Recalibration:**
   - Channel-wise scaling is applied to the input feature map.
   - Formula:
   $$\tilde{X}_c = s_c \cdot X_c$$
5. **Output:** A recalibrated feature map with the same dimensions as the input.

Squeeze-and-Excitation (SE) Networks



a block of squeeze-and-excitation networks
Zhao et al. *Artificial Intelligence Review* (2024)



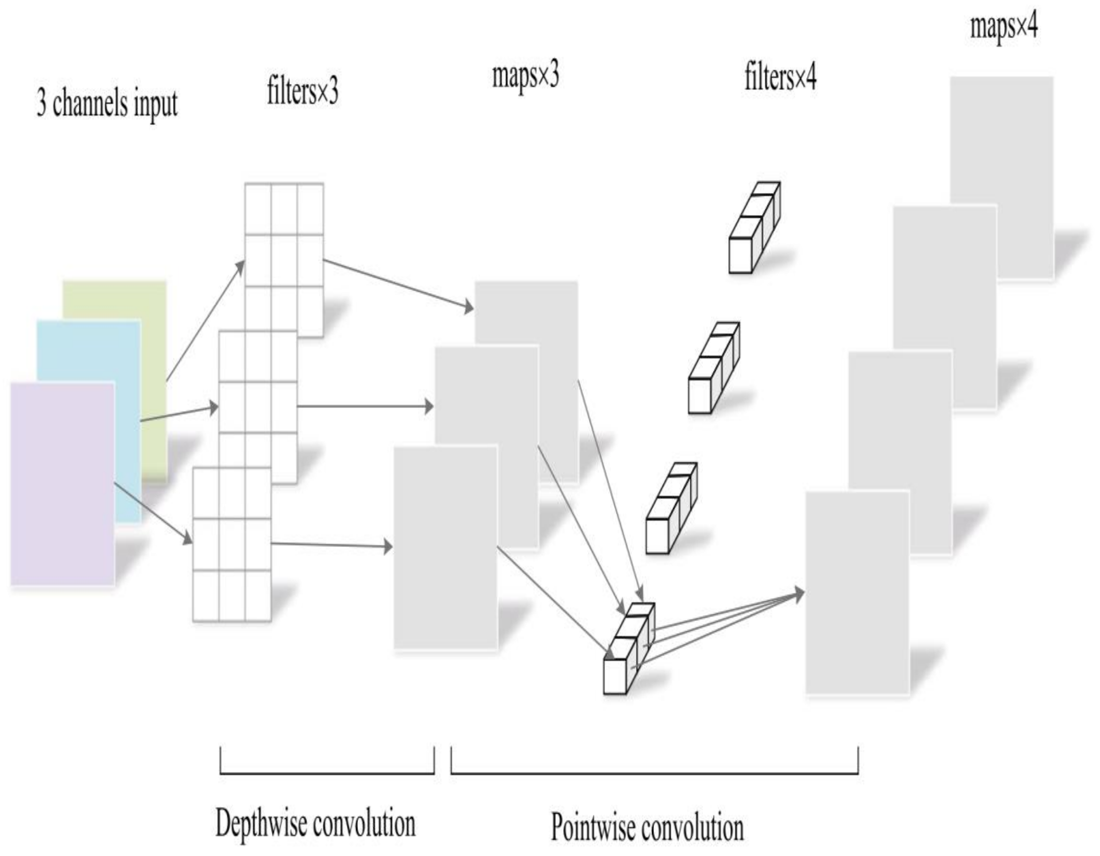
SENet Architecture
-- An Example Deep SE Network Structure for discriminating super- and typical enhancers by sequence information
Luo et al. *Computational Biology and Chemistry*, 2023

SENet

```
import torch
import torch.nn as nn
import torchvision.models as models

class SEBlock(nn.Module):
    def __init__(self, in_channels, reduction=16):

        super(SEBlock, self).__init__()
        self.global_avg_pool = nn.AdaptiveAvgPool2d(1)  # Global average pooling
        self.fc1 = nn.Linear(in_channels, in_channels // reduction, bias=False)  # Compression
        self.relu = nn.ReLU(inplace=True)
        self.fc2 = nn.Linear(in_channels // reduction, in_channels, bias=False)  # Expansion
        self.sigmoid = nn.Sigmoid()  # Scale factor

    def forward(self, x):
        batch_size, channels, _, _ = x.size()
        # Squeeze: Global average pooling
        y = self.global_avg_pool(x).view(batch_size, channels)
        # Excitation: Fully connected layers with ReLU and Sigmoid
        y = self.fc1(y)
        y = self.relu(y)
        y = self.fc2(y)
        y = self.sigmoid(y).view(batch_size, channels, 1, 1)
        # Scale: Multiply the input by the channel weights
        return x * y
```

MobileNet

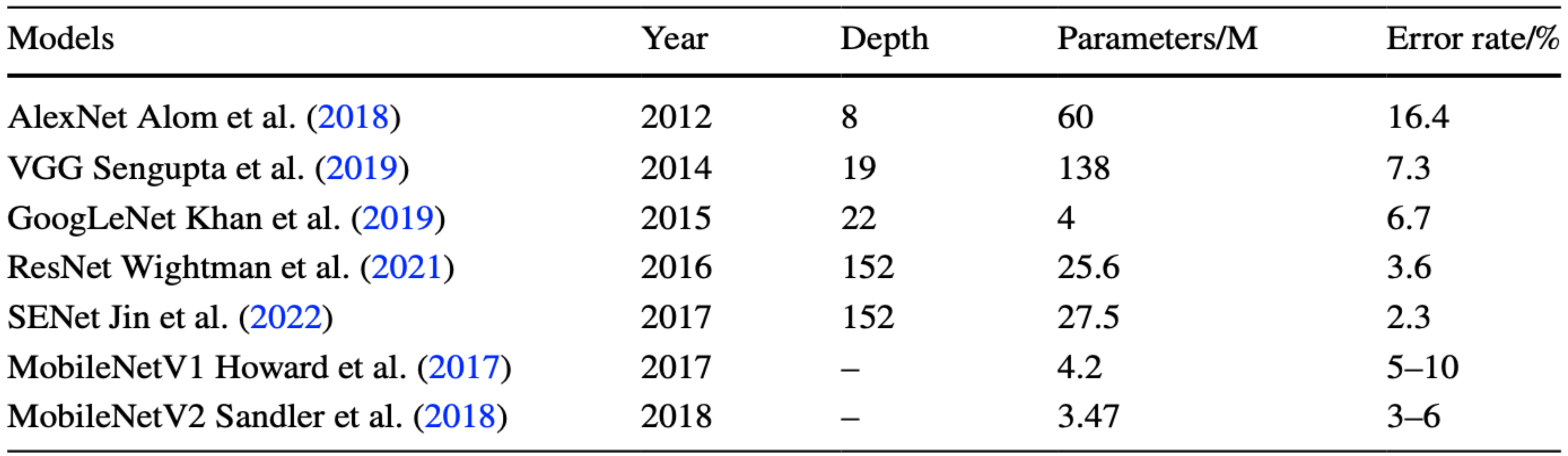
- **Motivation:** Traditional CNNs have large memory/computational demands, limiting deployment on mobile/embedded devices.
- **MobileNetV1 (Howard et al., 2017)** achieves near-VGG16 accuracy (~0.9% lower) with only 1/32 of its parameters.
  - **Key Technique:** Depthwise-separable convolutions (factorizing standard convolution into "depthwise" + "pointwise") → drastically fewer parameters and reduced FLOPs.
  - **Hyperparameters:** Width multiplier and resolution multiplier to balance accuracy and efficiency.
  - **Drawback:** Some kernels become zero during training → limited parameter utilization.



Architecture of MobileNetV1.
Howard et al. *arXiv* (2017)

Trends, Challenges, & Future Directions

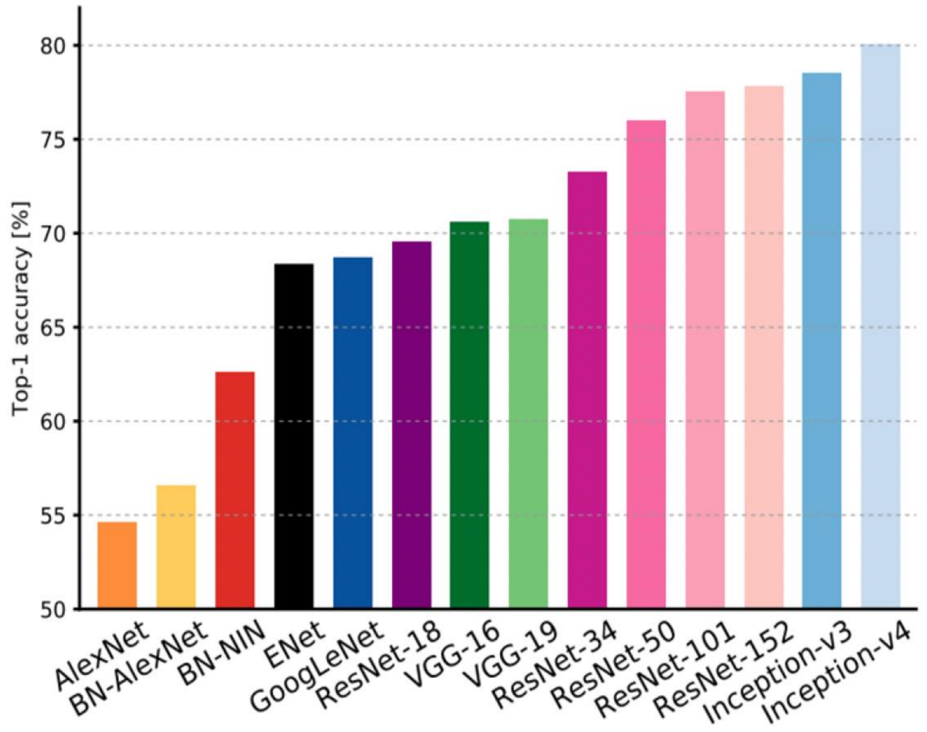
- Performance Comparisons of deep CNN networks show progressive accuracy improvements at varying parameter costs.

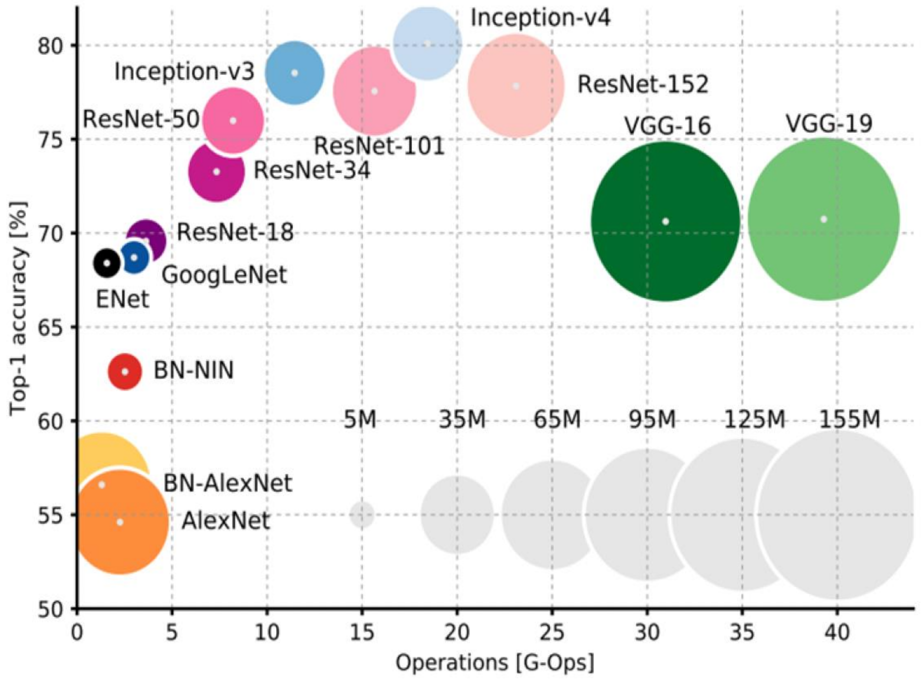


Models	Year	Depth	Parameters/M	Error rate/%
AlexNet Alom et al. (2018)	2012	8	60	16.4
VGG Sengupta et al. (2019)	2014	19	138	7.3
GoogLeNet Khan et al. (2019)	2015	22	4	6.7
ResNet Wightman et al. (2021)	2016	152	25.6	3.6
SENet Jin et al. (2022)	2017	152	27.5	2.3
MobileNetV1 Howard et al. (2017)	2017	–	4.2	5–10
MobileNetV2 Sandler et al. (2018)	2018	–	3.47	3–6

- **Challenges:**
  1. **Complexity & Resource Usage:** Advanced CNNs can be large and memory-intensive.
  2. **Data Dependence:** Labeled large-scale datasets are expensive and time-consuming to obtain.
  3. **Loss of Fine-grained Details:** Typical CNNs may struggle with small-sized inputs.

Comparing Complexity





An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Evaluation Metrics for Classification

Accuracy

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Precision

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

Recall (Sensitivity)

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

F1-Score

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Top-K Accuracy

$$\text{Top-K Accuracy} = \frac{\text{Number of Correct Predictions in Top-K}}{\text{Total Number of Predictions}}$$

TP, TN, FP, and FN represent true positives, true negatives, false positives, and false negatives, respectively.

Content

CNN Optimization Techniques

CNN optimization involves techniques to improve the performance, efficiency, and generalization of Convolutional Neural Networks during training and inference.

- **Goals**:

a) Reduce overfitting.

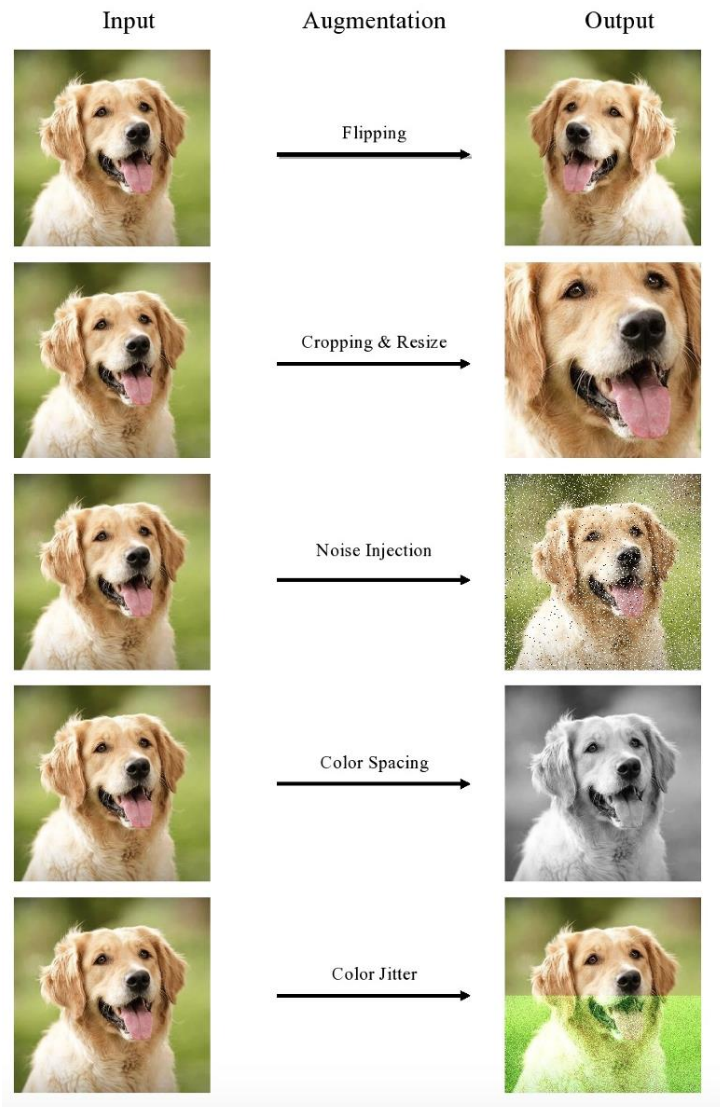
b) Improve convergence speed.

c) Optimize computational resources.

- **Common Strategies:**
    a) Data Augmentation
    b) Regularization (L1, L2, Elastic Net)
    c) Dropout & Early Stopping
    d) Hyperparameter Optimization
    e) Transfer Learning

Data Augmentation

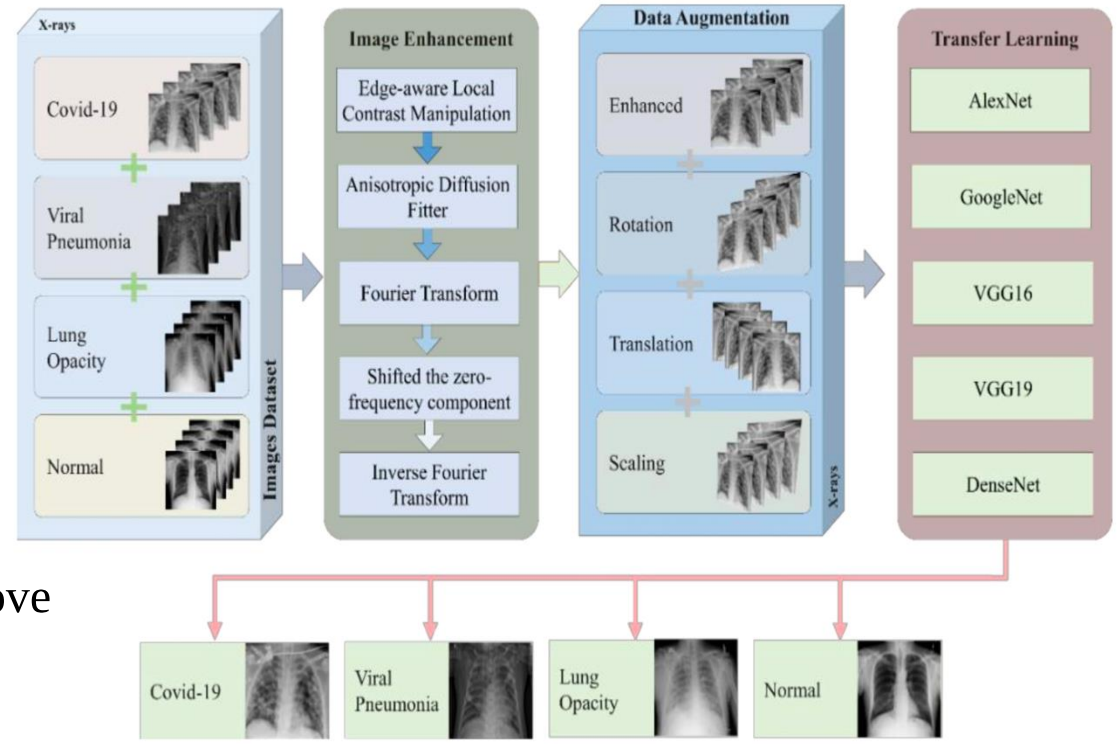
- Data augmentation is a strategy used to artificially increase the size and diversity of a training dataset by applying transformations to the existing data.
- **Purpose:** Improve model generalization. Prevent overfitting. Compensate for limited training data.
- *Why?* Increases effective training set size without extra data collection.
- **Common Methods:**
  - Color jittering, cropping, flipping, rotations, scaling.
  - PCA-based color augmentation (as in AlexNet) (Krizhevsky et al., *Commun. ACM*, 2017).
- Transfer learning approach using well-known CNN models (GoogleNet, AlexNet, VGG16, VGG19, DenseNet, etc.) along with data augmentation techniques can be used to accelerate the training and testing process while yielding good results and performance.
- He et al. implemented data augmentation along with regularization techniques such as dropouts and weight decay (CVPR, 2016).



Teerath et al. *IEEE Access* (2024)

Data Augmentation

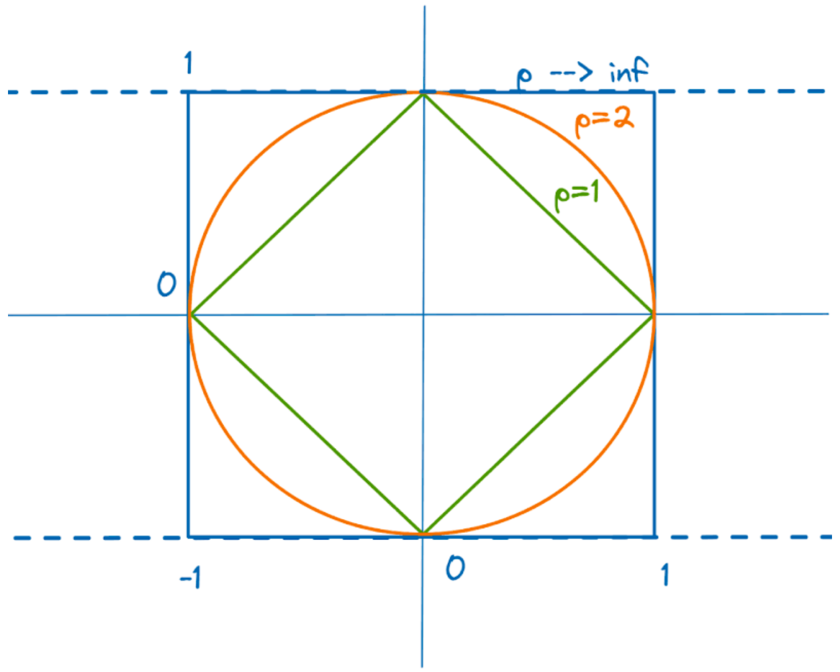
- **Geometric Transformations:**
  Flipping: Horizontal and vertical flips.
  Rotation: Rotates images by a specified angle.
  Scaling: Resizes images while preserving aspect ratio.
  Cropping: Extracts subregions from the image.
- **Color Transformations:**
  Brightness Adjustment: Alters image brightness.
  Contrast Adjustment: Modifies contrast levels.
  Saturation Adjustment: Changes color saturation.
  Hue Adjustment: Shifts color hues.
- **Noise Injection:** Adds random noise to images to improve robustness.
- **Affine Transformations:** Applies scaling, shearing, or translation to the images.



Example of using preprocessing techniques along with the well-known CNN models for COVID-19 and Lungs Pneumonia detection using transfer learning.
Latif et al. *AIMS Mathematics* (2024)

Regularization methods

- **Definition**: Regularization refers to techniques that improve a model's generalization by reducing overfitting to the training data.
- **Why Regularization?** Deep learning models are prone to overfitting due to high capacity and complex structures. Regularization helps balance the trade-off between model complexity and performance.
- **L2 Regularization (Weight Decay)**
  - Penalizes the square of weights → discourages large weight values, helps smooth solutions.
- **L1 Regularization (Lasso)**
  - Penalizes the absolute value of weights → encourages sparsity (some weights become zero).
- **Elastic Net**
  - Combines L1 and L2 → can both shrink weights and promote sparsity.

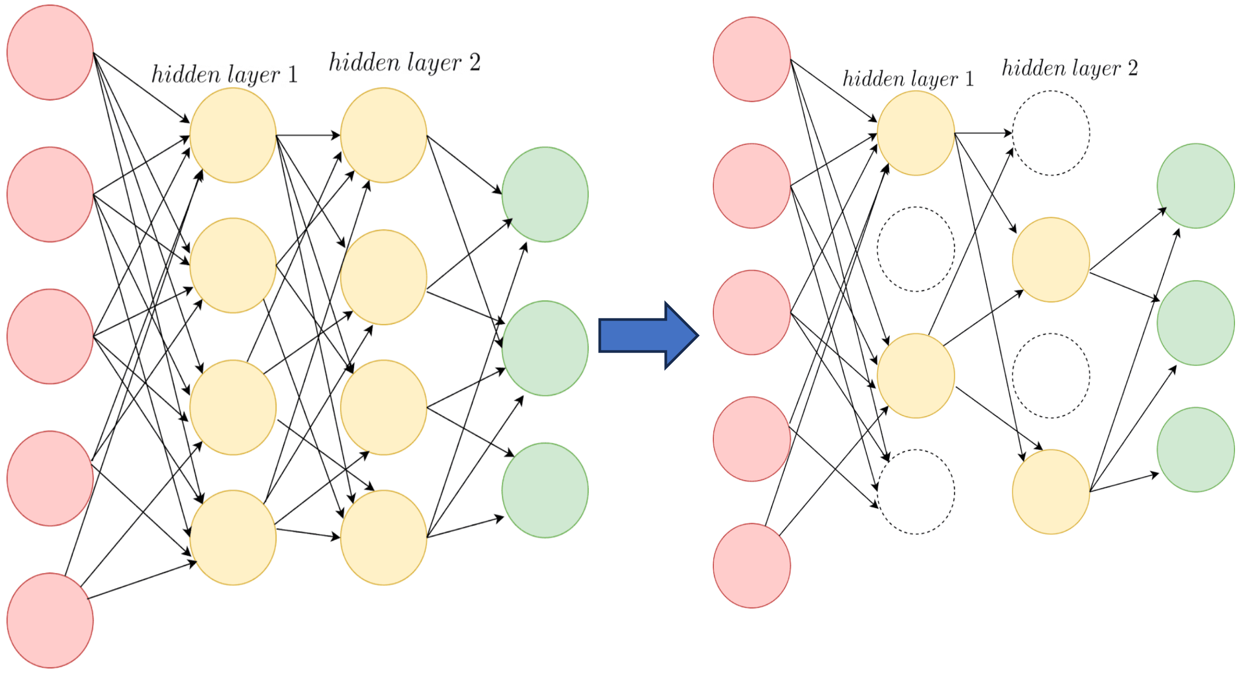


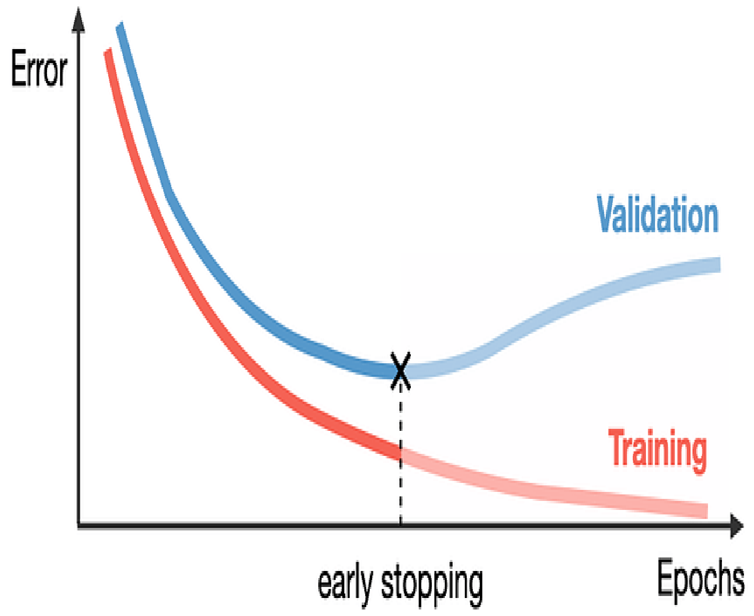
Dropout & Early Stopping

- **Dropout**
  - Randomly "drops" neurons during training.
  - Reduces co-adaptations among neurons → mitigates overfitting.

- **Early Stopping**

  Monitors validation performance and halts training before overfitting sets in. Balances bias/variance by stopping at the optimal point.

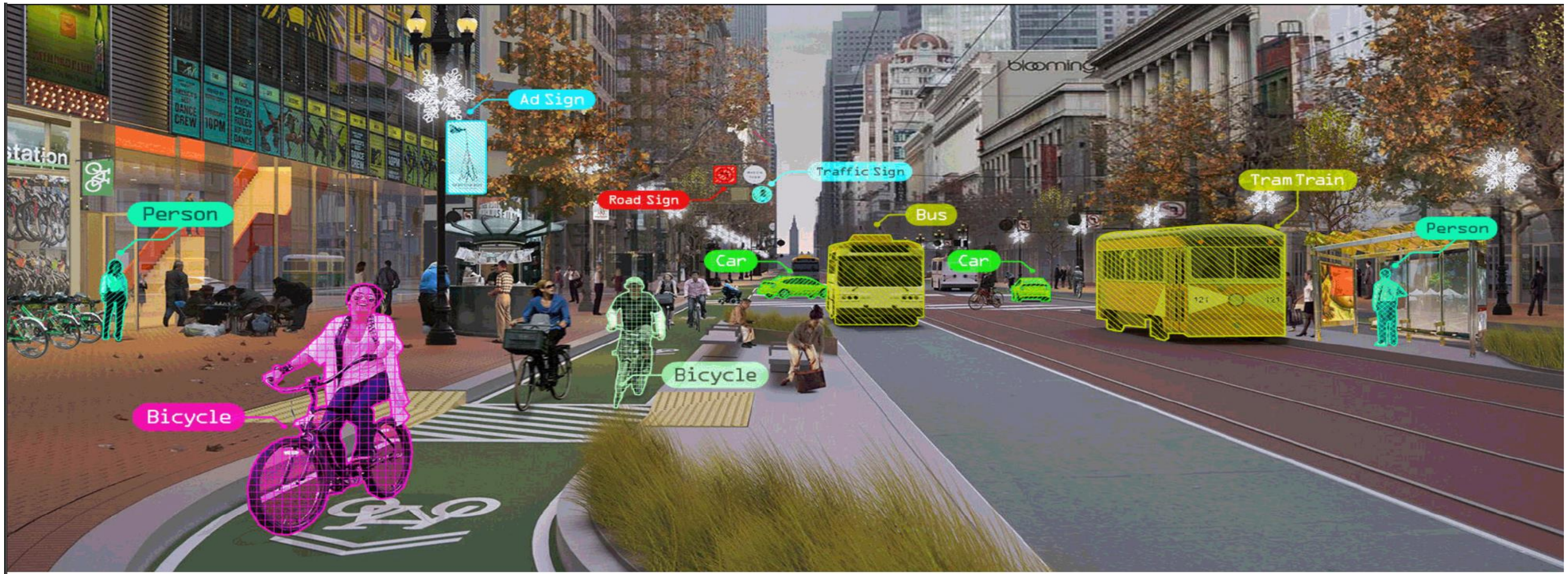




https://www.pinecone.io/learn/regularization-in-neural-networks/ https://www.comet.com/site/blog/4-techniques-to-tackle-overfitting-in-deep-neural-networks/

Content

Object Detection



Challenge:

- Objects can be anywhere in the scene, in any orientation, rotation, color hue, etc.
- How can we overcome this challenge?

Answer:

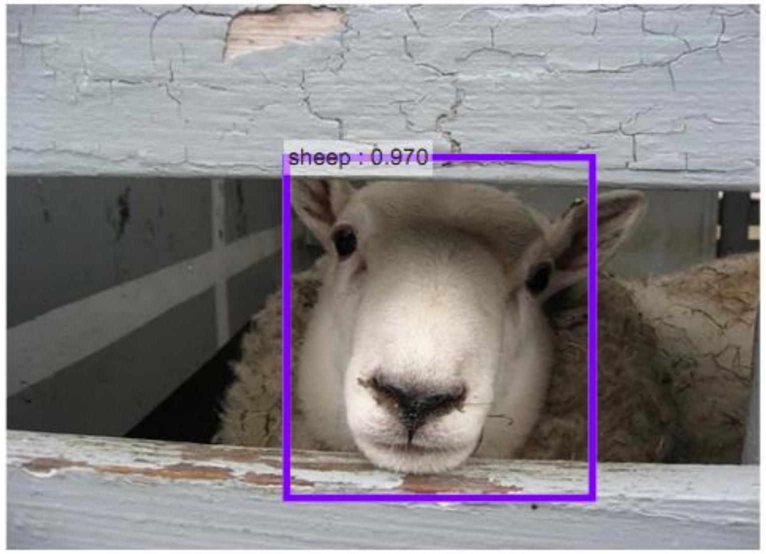
- Learn a ton of features (millions) from the bottom up
- Learn the convolutional filters, rather than pre-computing them

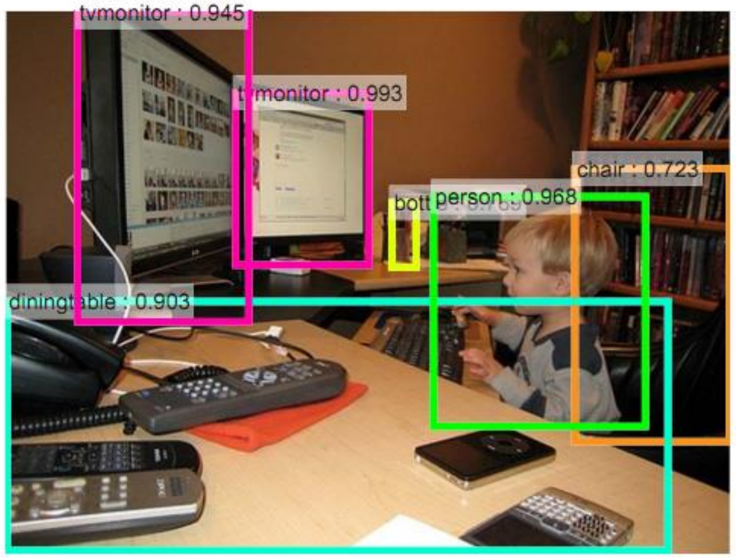
What is Object Detection?

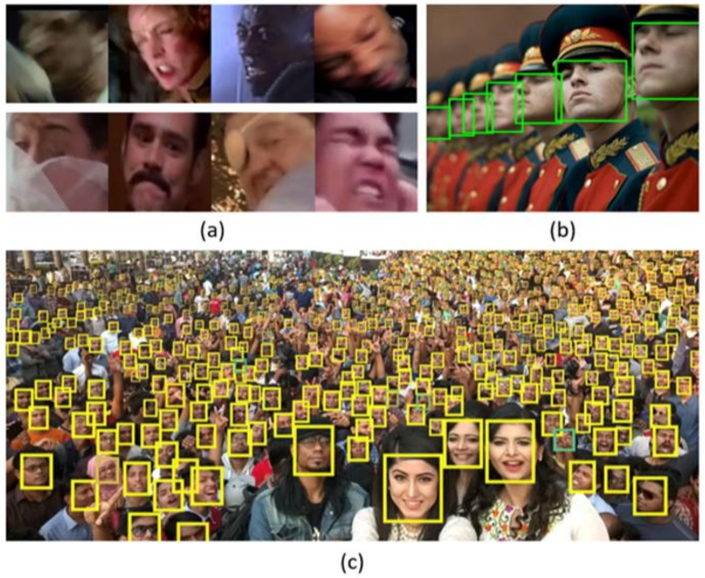
To determine: *What objects are where?*

-- Object bounding box: location and size

-- Object category.

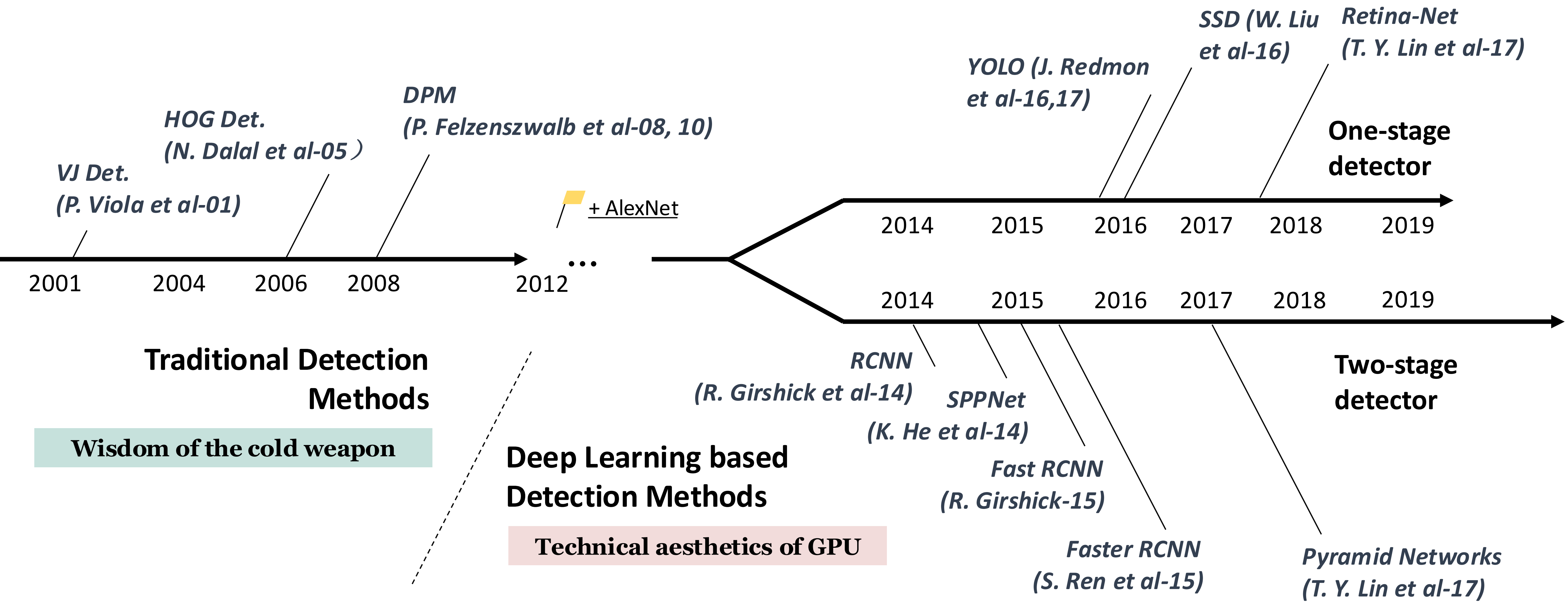






By NIPS15-Faster RCNN

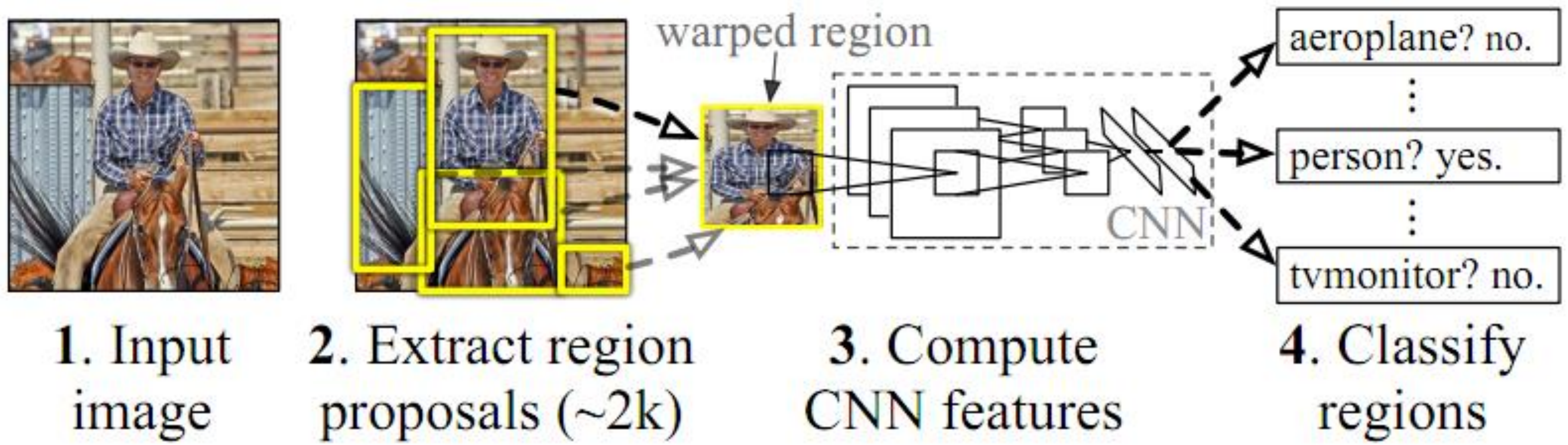
Object Detection Milestones



VJ Det.
(P. Viola et al-01)
HOG Det.
(N. Dalal et al-05）
DPM
(P. Felzenszwalb et al-08, 10)
+ AlexNet
2001
2004
2006
2008
2012
...
Traditional Detection Methods
Wisdom of the cold weapon
Deep Learning based Detection Methods
Technical aesthetics of GPU
YOLO (J. Redmon et al-16,17)
SSD (W. Liu et al-16)
Retina-Net (T. Y. Lin et al-17)
One-stage detector
2014
2015
2016
2017
2018
2019
2014
2015
2016
2017
2018
2019
RCNN (R. Girshick et al-14)
SPPNet (K. He et al-14)
Fast RCNN (R. Girshick-15)
Faster RCNN (S. Ren et al-15)
Pyramid Networks (T. Y. Lin et al-17)
Two-stage detector

R-CNN: Regions with CNN features

Ross B. Girshick et al., (CVPR2014)



- Object Proposal+CNN features
- Bounding Box Regression
- Fine tuning
- VOC07 mAP: 33.7→58.5

Time: 14s/image on a GPU

Drawbacks

- The redundant feature computations on a large number of overlapped proposals (>2000 boxes/img) leads to an extremely slow detection speed (14s per image with GPU).

R-CNN: Regions with CNN features

Definition: R-CNN is a deep learning framework for object detection introduced by Ross Girshick in 2014. It integrates region proposals with CNNs to detect objects in an image effectively.

Key Contributions:

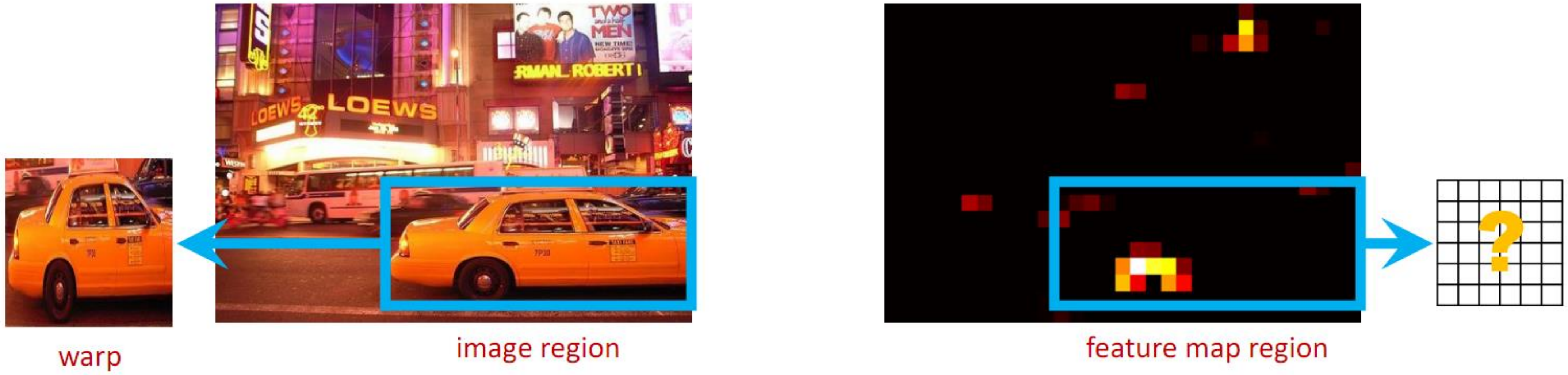
- ❖ Combines region proposals with CNN-based feature extraction.
- ❖ Demonstrates the use of transfer learning for detection tasks.
- ❖ Achieves significant performance improvements over traditional methods.

Workflow of R-CNN:

- ➢ Input image is processed using Selective Search to generate region proposals.
- ➢ Each region is resized to 224x224 and passed through a CNN to extract features.
- ➢ SVM classifiers predict object categories for the proposals.
- ➢ Bounding box regression refines the coordinates of the proposals.
- ➢ Outputs are the predicted class labels and refined bounding boxes.

SPPNet: Spatial Pyramid Pooling

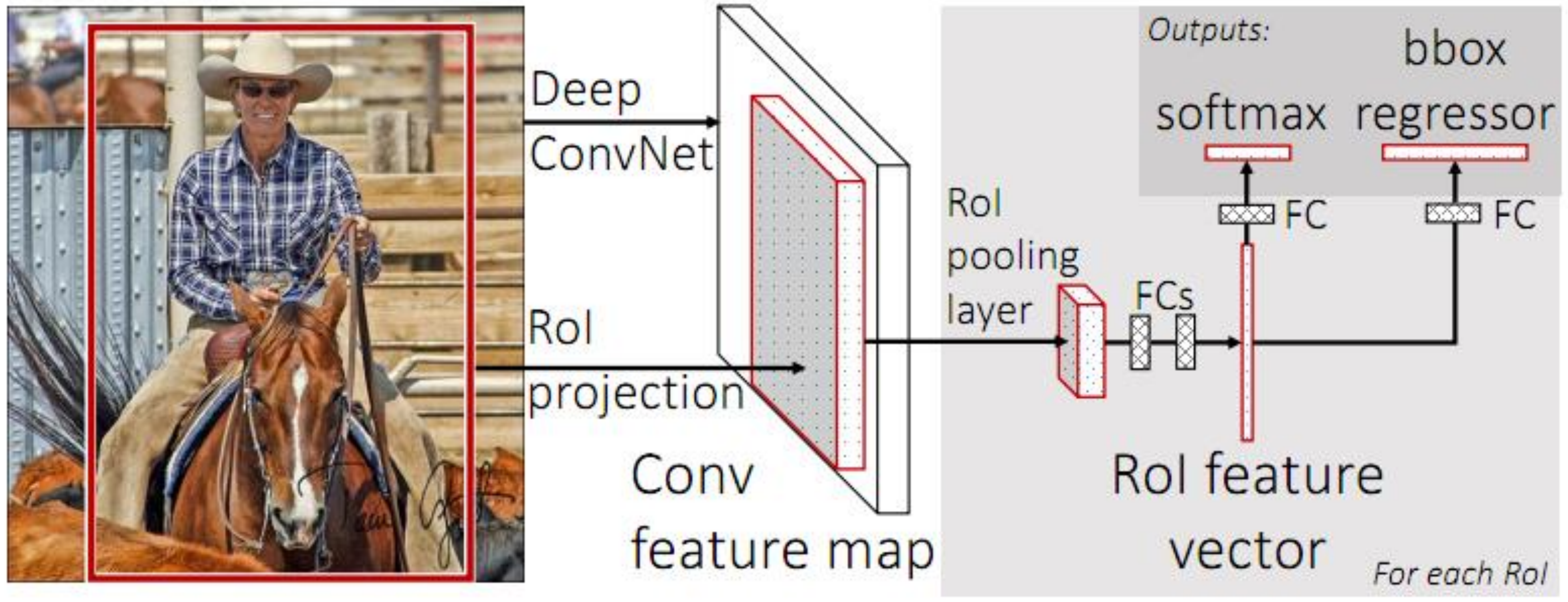
SPPnet is a deep learning framework designed to handle images of arbitrary sizes without requiring cropping or resizing. It introduces the Spatial Pyramid Pooling (SPP) layer, which allows for flexible input dimensions and improved computational efficiency.



- Fixed-length features are required by fully-connected layers or SVM
- But how to produce a fixed-length feature from a feature map region?
- Solutions in traditional compute vision: Bag-of-words, SPM...

Kaiming He et al., (ECCV2014)

Fast RCNN



Fast R-CNN is an object detection framework introduced by Ross Girshick in 2015. It improves upon the inefficiencies of R-CNN by introducing Region of Interest (ROI) Pooling and enabling shared computation, leading to faster and more accurate object detection.

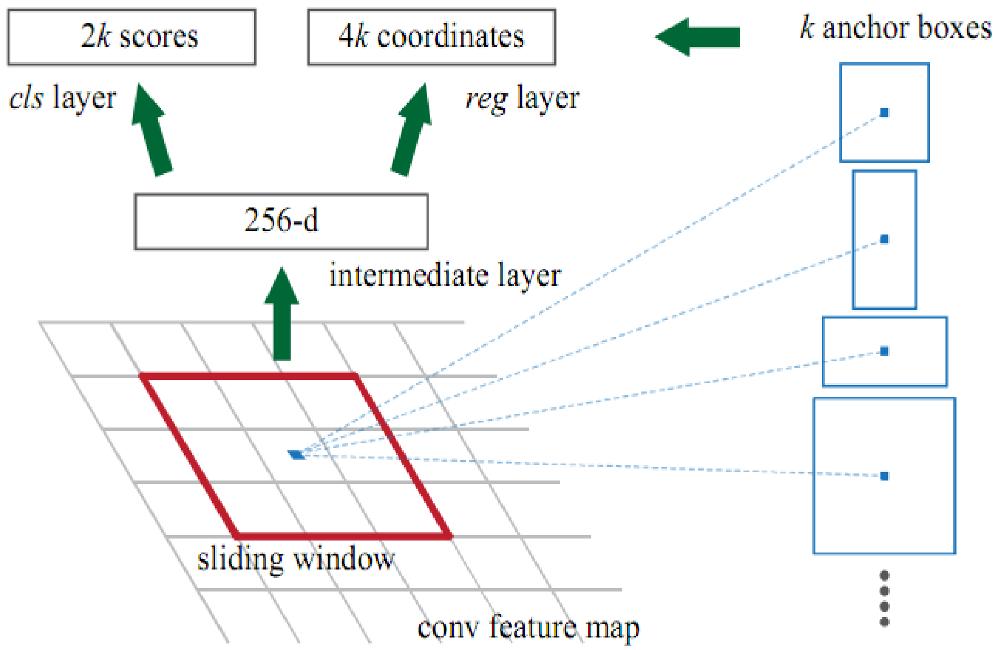
- ROI Pooling
- Multi-task loss (Clc. + BB Reg.)
- BP through RoI pooling layers
- VOC07 mAP: 58.5→70.0

Time:
0.32s/image on a GPU

Ross B. Girshick (ICCV15)

Faster RCNN

Faster R-CNN is a successor to Fast R-CNN and introduces the **Region Proposal Network (RPN)** for generating region proposals, making the detection pipeline fully end-to-end.



Anchors (reference boxes)

- Region Proposal Network
- Detection Network
- Sharing Features
- VOC07 mAP: 70.0→78.8

Time: 17 fps on a GPU

Shaoqing Ren et al., (NIPS2015)

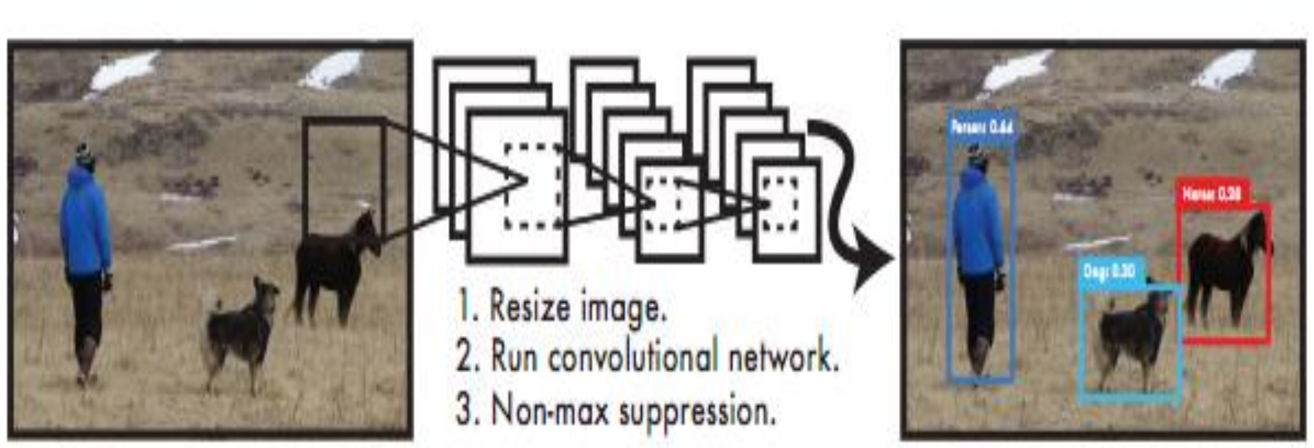
$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i, t_i^*).$$

You Only Look Once (YOLO)

YOLO treats object detection as a single regression problem, predicting both class probabilities and bounding box coordinates in one forward pass. *J. Redmon et al., (CVPR2016)*

Key Contributions:

- ❖ Introduces a unified framework for object detection, enabling real-time performance.
- ❖ Processes the entire image in a single forward pass, improving efficiency.
- ❖ Balances speed and accuracy, making it suitable for real-world applications.



- Runs at 45fps with VOC07 mAP=63.4% and VOC12 mAP=57.9%.
- A fast version runs at 155fps with VOC07 mAP=52.7%.

Workflow of YOLO

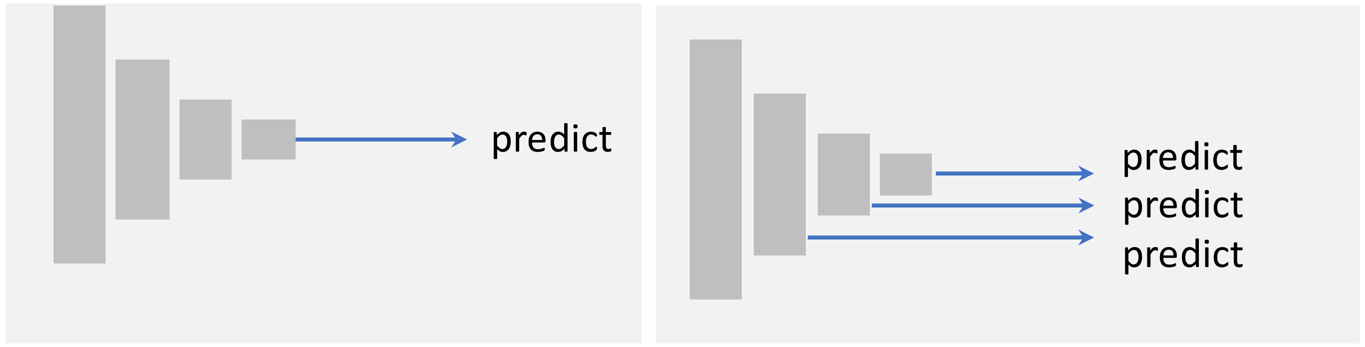
- **Input Image:** The input image is divided into an SXS grid (e.g., 7X7).
- **Feature Extraction:** A CNN processes the image to extract features.
- **Bounding Box Prediction:** Each grid cell predicts:
  - Bounding boxes (coordinates and dimensions).
  - Confidence scores for each bounding box.
- **Classification:** Each grid cell predicts class probabilities for the objects it contains.
- **Post-Processing:** Non-Maximum Suppression (NMS) removes duplicate detections and retains the most confident predictions.

SSD: Single Shot MultiBox Detector

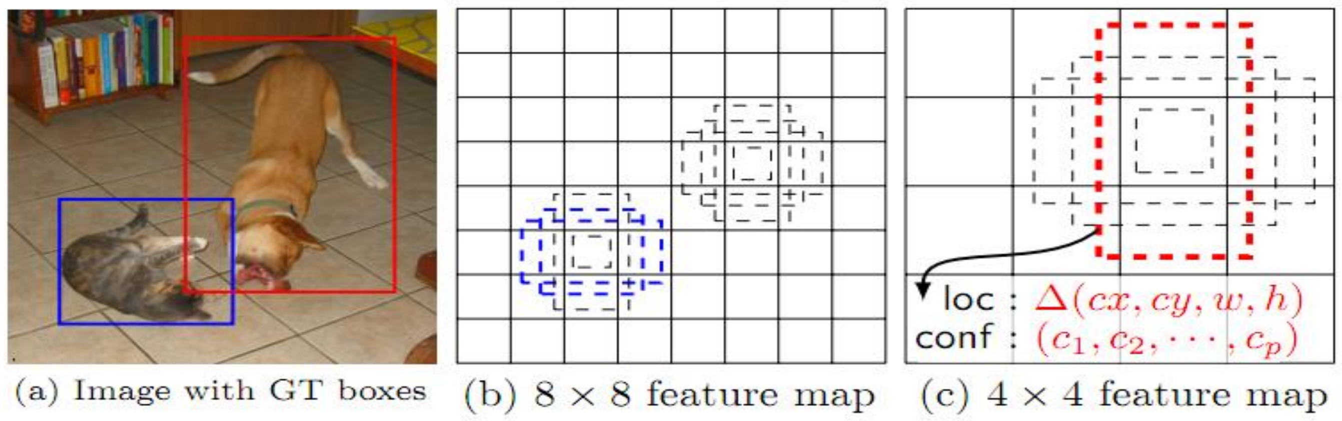
SSD performs object detection in a single forward pass, making it fast and efficient compared to region-based methods like Faster R-CNN.

Key Contributions:

- Uses multi-scale feature maps for detecting objects of different sizes.
- Introduces default (prior) boxes for efficient bounding box predictions.
- Eliminates the need for separate region proposal steps, improving speed.



(a) YOLO (b) SSD

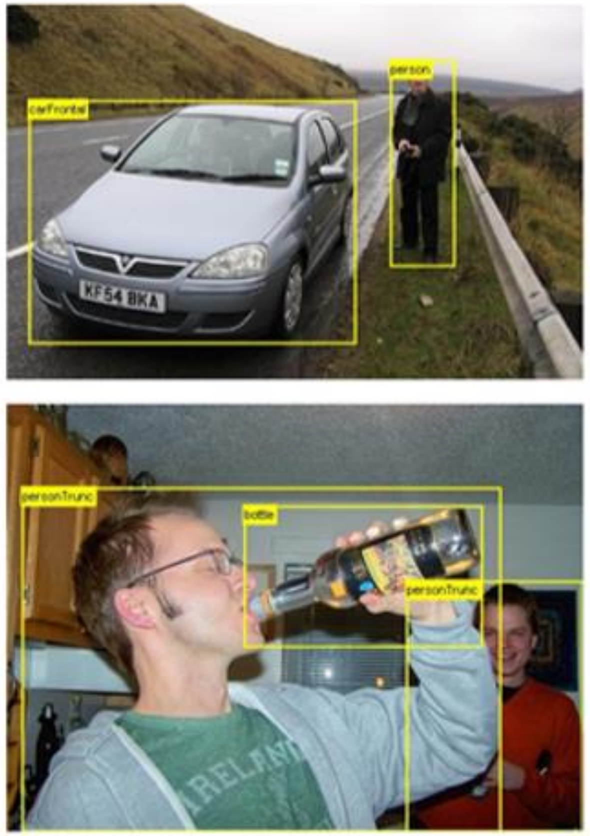

loc : $\Delta(cx, cy, w, h)$
conf : $(c_1, c_2, \cdots, c_p)$

(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

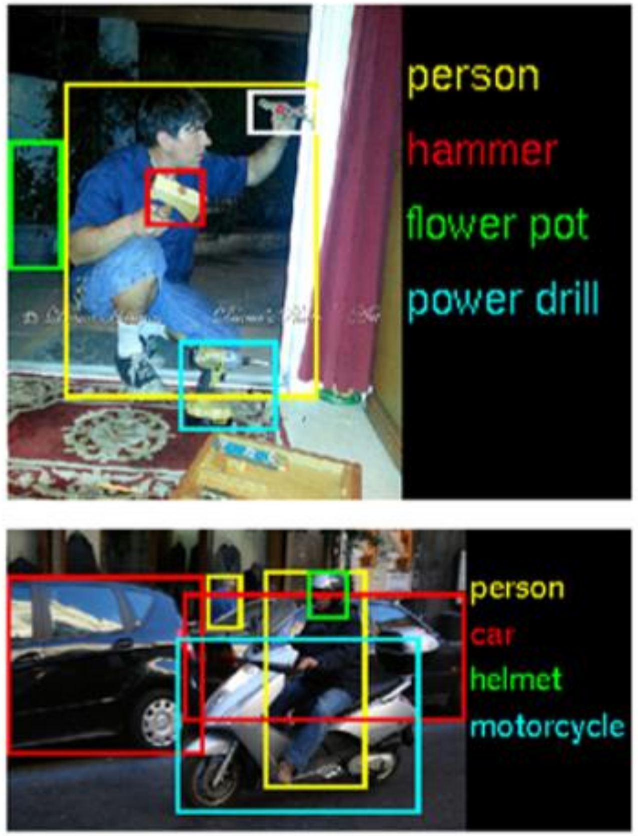
- Multi-resolution detec.
- Multi-reference detec. (anchor boxes)
- Hard negative mining
- VOC07 mAP=76.8%, VOC12 mAP=74.9%.
- The fast version runs at 59fps.

Wei Liu et al., (ECCV2016)

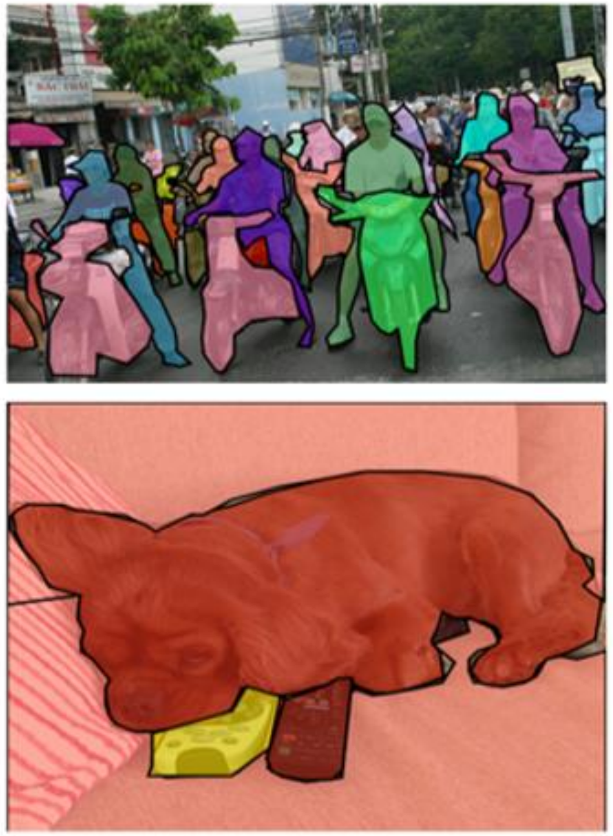
Detection Datasets



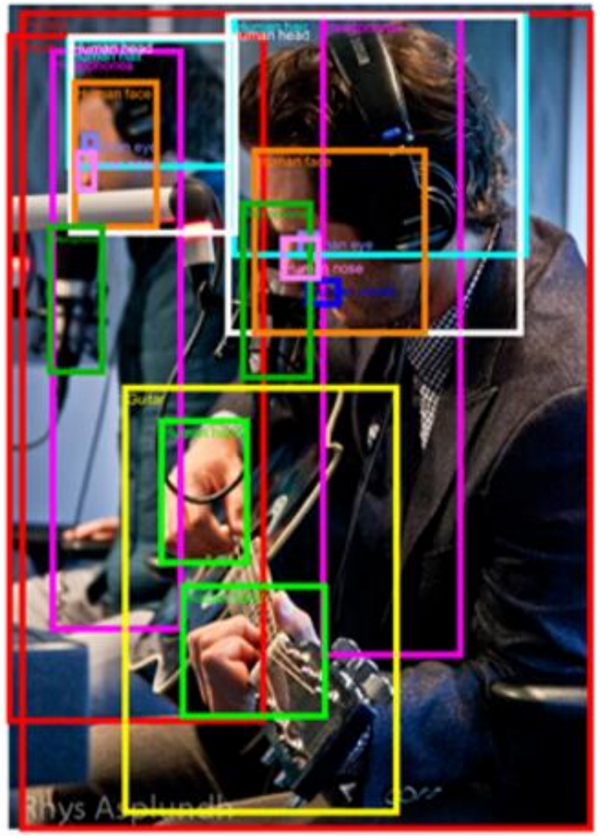
PASCAL VOC



ILSVRC



MS-COCO



Open Images

Detection Datasets

Dataset	train		validation		trainval		test	
	images	objects	images	objects	images	objects	images	objects
VOC-2007	2,501	6,301	2,510	6,307	5,011	12,608	4,952	14,976
VOC-2012	5,717	13,609	5,823	13,841	11,540	27,450	10,991	-
ILSVRC-2014	456,567	478,807	20,121	55,502	476,688	534,309	40,152	-
ILSVRC-2017	456,567	478,807	20,121	55,502	476,688	534,309	65,500	-
MS-COCO-2015	82,783	604,907	40,504	291,875	123,287	896,782	81,434	-
MS-COCO-2018	118,287	860,001	5,000	36,781	123,287	896,782	40,670	-
OID-2018	1,743,042	14,610,229	41,620	204,621	1,784,662	14,814,850	125,436	625,282

TABLE 1
Some well-known object detection datasets and their statistics.

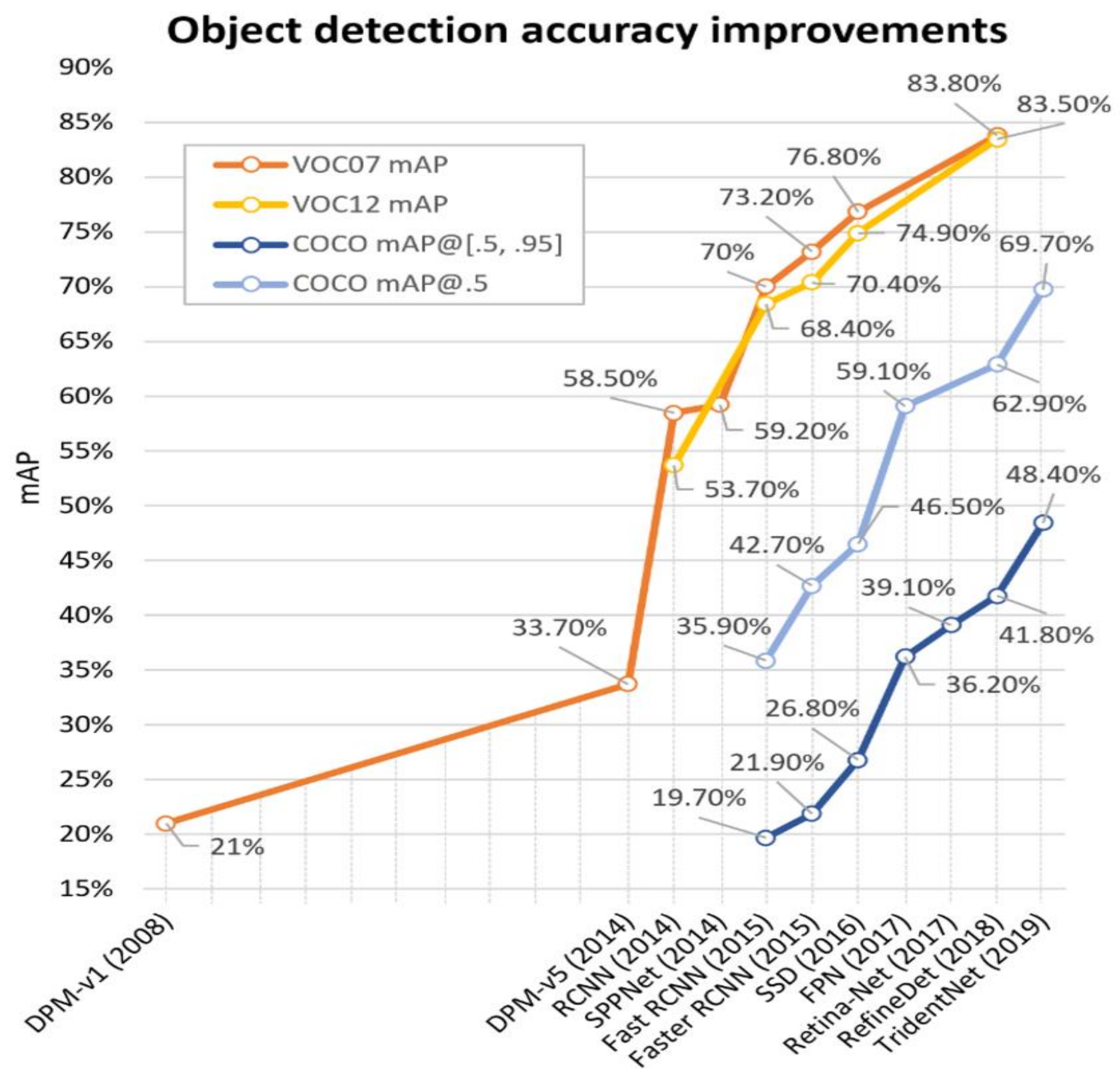
Detection accuracy improvement

VOC07 Train+val

- 5,011 imgs
- 12,608 objs
- 20 classes

MSCOCO Train+val

- 123,287 imgs
- 896,782 objs
- 80 classes



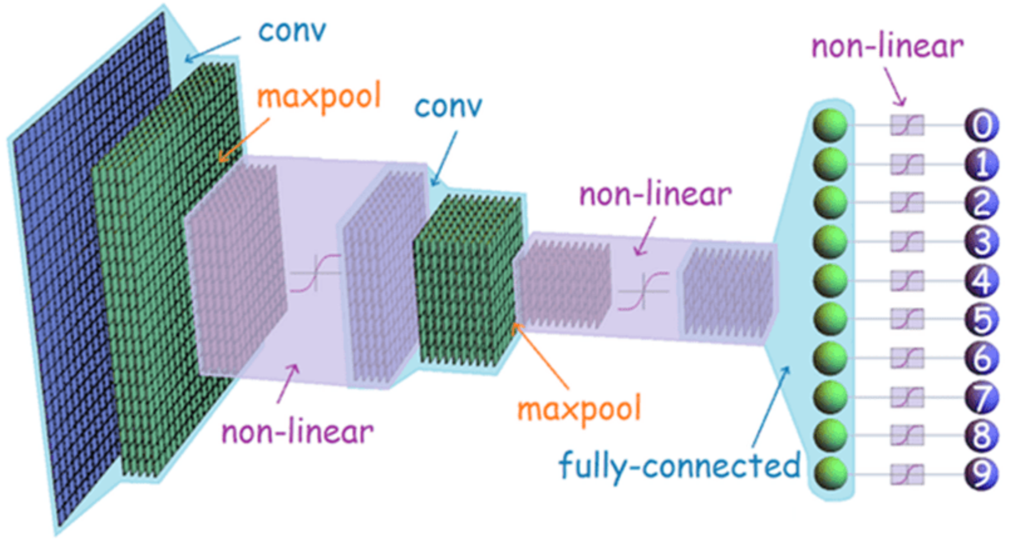
3D CNN

- **Focus on volumetric or temporal information**

  3D Convolutional Neural Networks (3D CNNs) extend the functionality of 2D CNNs by operating on three-dimensional input data.

  While 2D CNNs process spatial information in two dimensions (height and width), 3D CNNs add a third dimension, allowing them to capture volumetric or temporal information.

- **Overview of standard applications (e.g., video sequences or 3D medical imaging)**



A 3D CNN architecture. (source: Handwritten Digit Recognition Using CNN with Keras)

Difference Between 2D and 3D CNNs

- **2D CNNs:** Apply convolution on 2D data (e.g., images) using 2D filters of size n×n. The input is a matrix of shape m×m×r, where m is the spatial dimension and r is the number of channels. The output is a set of 2D feature maps. The convolution operation in 2D CNN is defined as:

$$(f * x)(i,j) = \sum_{m}\sum_{n} x(i+m, j+n) f(m,n)$$

  where f is the filter, x is the input, and $*$ represents the convolution.

- **3D CNNs:** Perform convolution across three dimensions (height, width, and depth/temporal axis) with 3D filters of size n×n×n. The input is a 3D tensor, m×m×m×r, capturing spatial and depth/temporal dimensions. The output consists of 3D feature maps, providing deeper feature representations. The convolution operation in 3D CNN is defined as:

$$(f * x)(i,j,k) = \sum_{m}\sum_{n}\sum_{o} x(i+m, j+n, k+o) f(m,n,o)$$

  where f is the 3D filter, x is the 3D input, and $*$ represents the 3D convolution.

Biomedical Applications for 3D CNNs

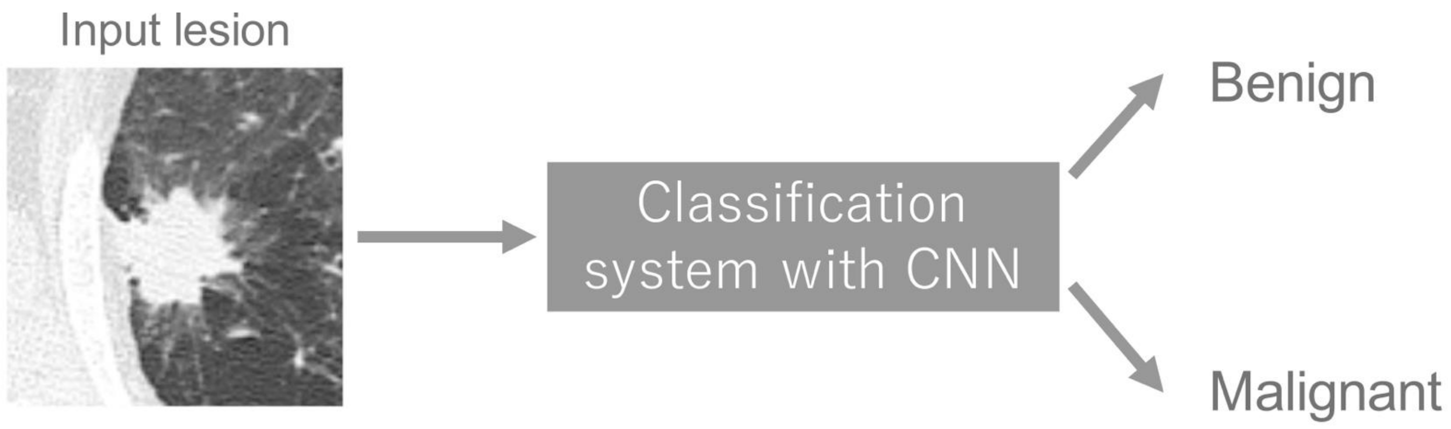
- **Medical Imaging**: 3D CNNs are highly effective for analyzing volumetric data from medical imaging modalities, such as MRI, CT scans, or PET scans, where the input data are three-dimensional. For example, in brain MRI, a 3D CNN can capture the spatial relationships between different brain regions and predict clinical outcomes.

    - **Alzheimer's Disease Prediction**: By using 3D CNNs, researchers can analyze 3D brain scans to detect subtle structural changes associated with Alzheimer's disease (AD).

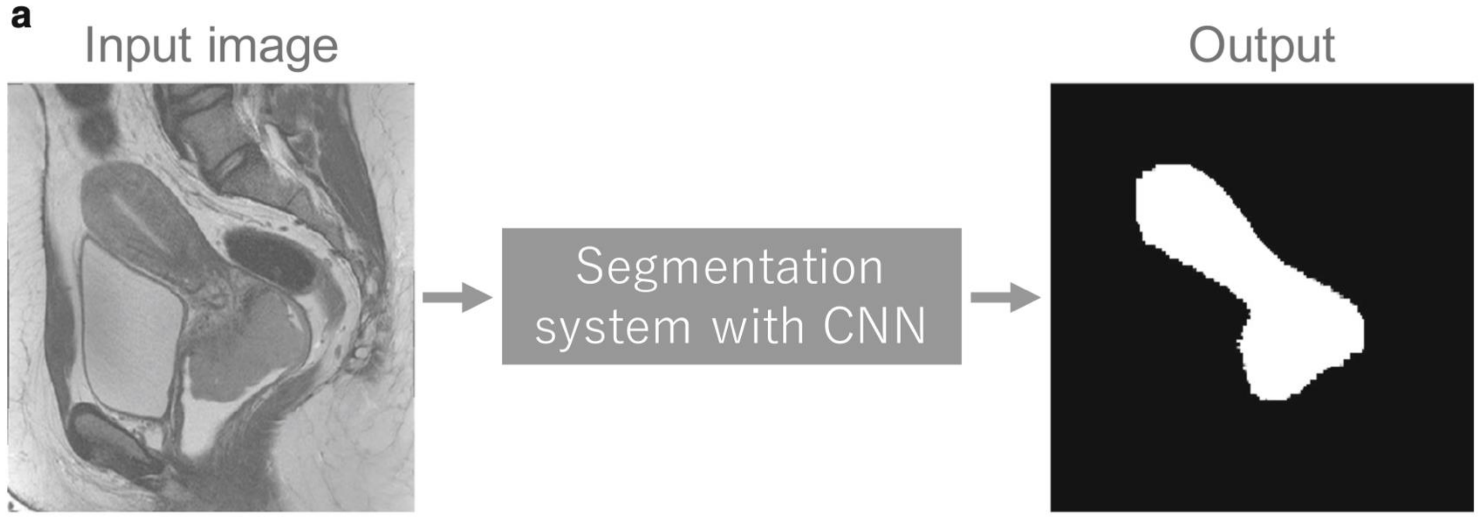
    - **Tumor Detection**: In 3D CT or MRI scans, 3D CNNs can identify tumors by learning volumetric patterns within the body, aiding in early diagnosis and treatment planning.

- **Functional Connectivity Analysis**: In neuroscience, 3D CNNs are employed to analyze 4D functional MRI data (3D + time), helping to map brain connectivity and investigate conditions such as autism or schizophrenia.

Biomedical Applications for 3D CNNs



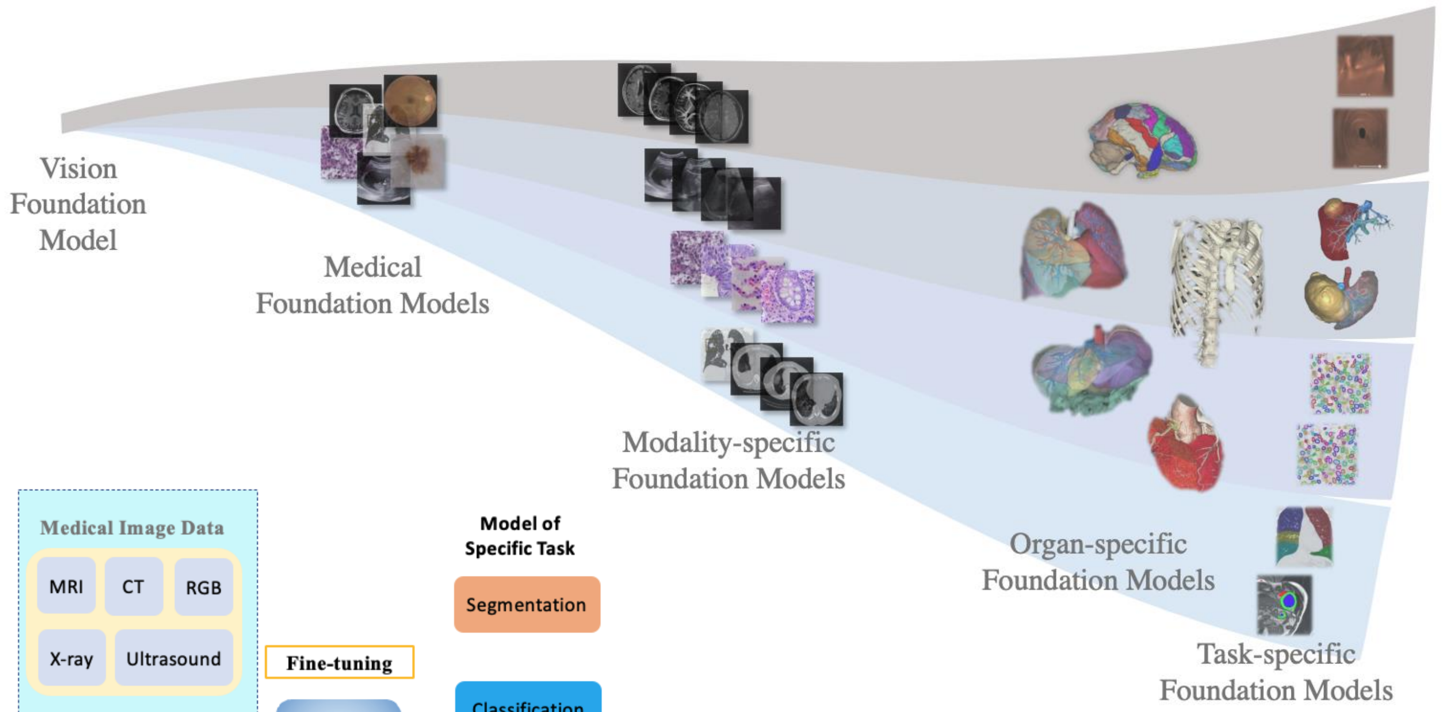
A schematic illustration of a tumor classification system with CNN
Yamashita et al. *Insights into Imaging* (2018)

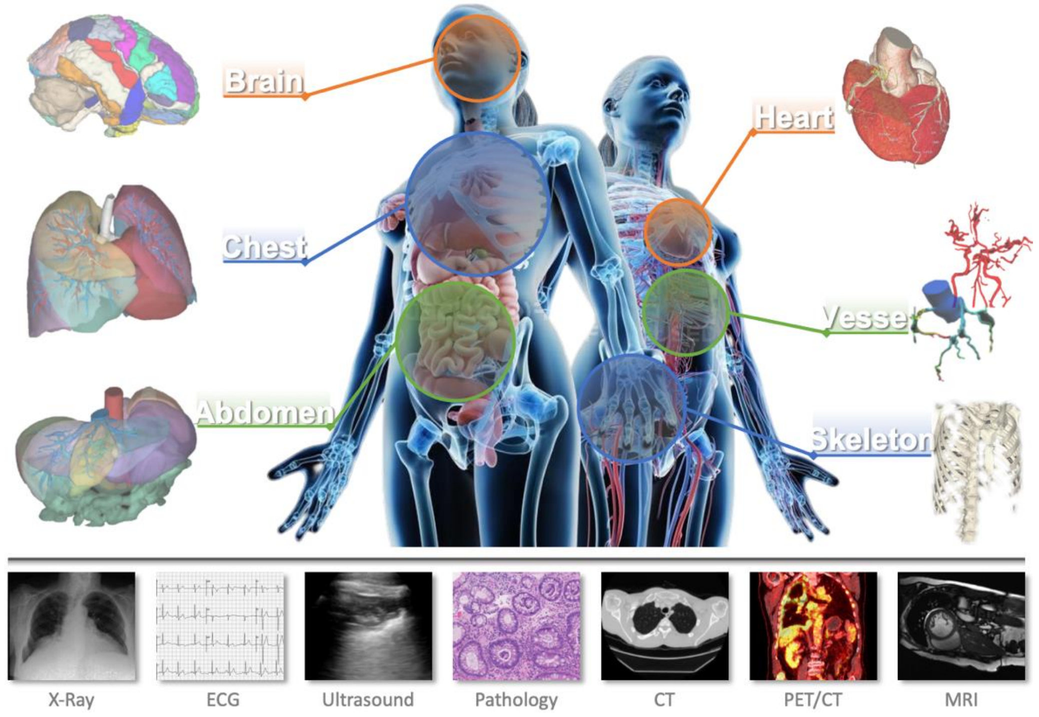


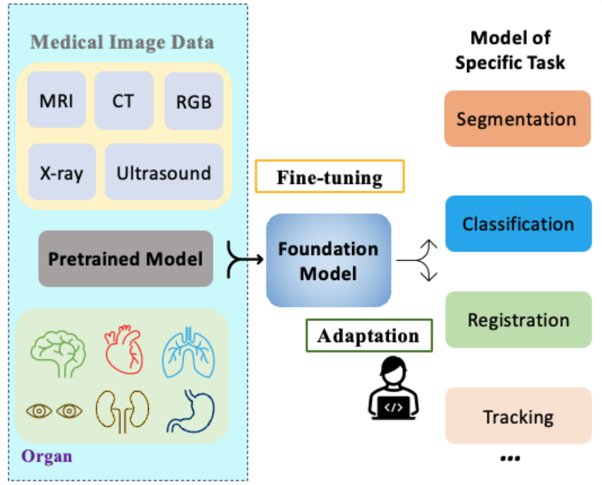
A schematic illustration of a tumor segmentation system with CNN
Yamashita et al. *Insights into Imaging* (2018)

Medical Image Foundation Model (MIFM)

Spectrum of foundation models in medical image analysis


Vision Foundation Model
Medical Foundation Models
Modality-specific Foundation Models
Organ-specific Foundation Models
Task-specific Foundation Models

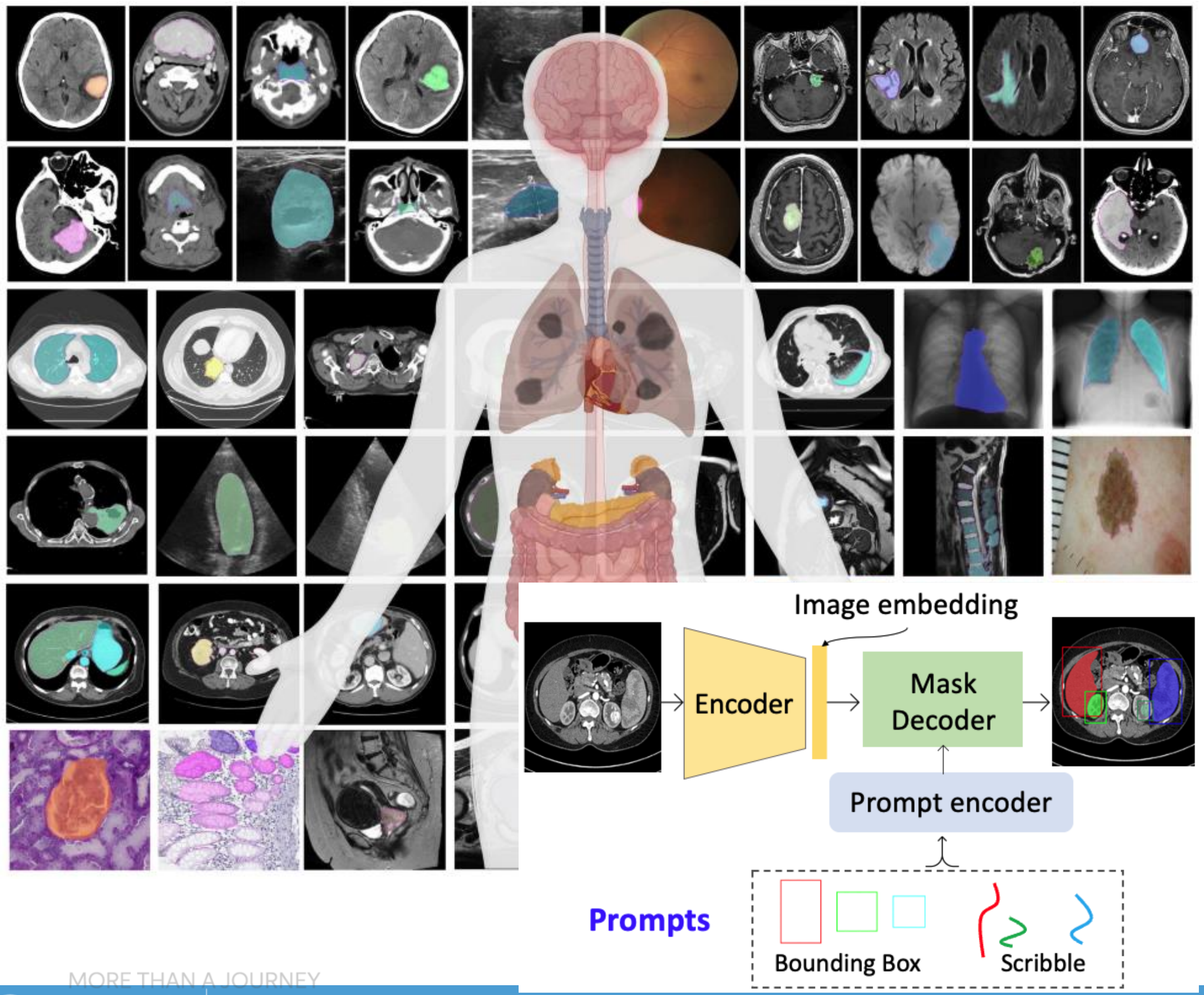




1) Medical foundation models have immense potential in solving a wide range of downstream tasks
2) Help to accelerate the development of accurate and robust models, reduce the dependence on large amounts of labeled data

MIFM for Segmentation

MedSAM: Segment Anything in Medical Images



1) Developed on a large-scale medical image dataset with 1,570,263 image-mask pairs, covering 10 imaging modalities and over 30 cancer types.

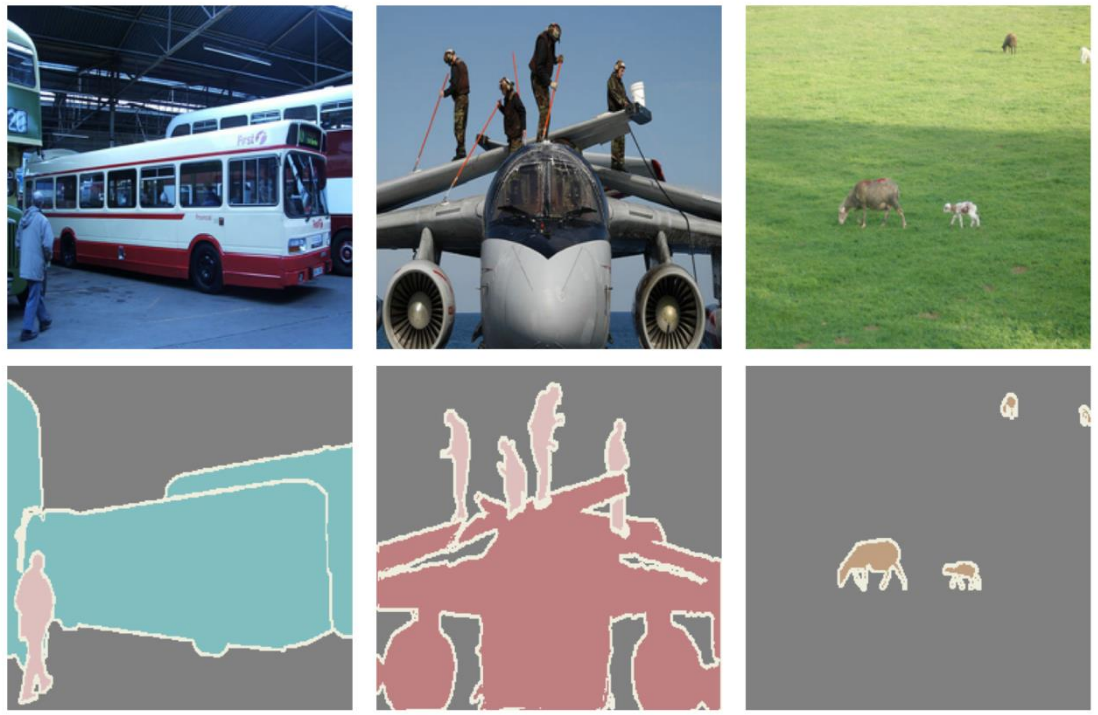
2) Evaluation on 86 internal validation tasks and 60 external validation tasks, demonstrating better accuracy and robustness than modality-wise specialist models.

3) Delivering accurate and efficient segmentation across a wide spectrum of tasks.

Content

Statistical Generative Models

$$X = X_{\text{obj}}^1(u_1) \oplus \cdots \oplus X_{\text{obj}}^J(u_J) \oplus \epsilon, \quad J \sim \text{Poisson}(\Lambda(\Omega)), \quad u_j \sim P_\lambda, \quad j = 1, \ldots, J.$$



(a) A man and three buses. (b) Four men and an airplane. (c) Four cows.

Figure 1: Sample images from the VOC2012 dataset. The top row displays original images, while the bottom row highlights the objects with the background shaded in gray.

$$Y = F^*(\rho(X_{\text{obj}}, H_1), \ldots, \rho(X_{\text{obj}}, H_{M^*})),$$

- ❖ Object Size
- ❖ The Number of Objects
- ❖ Spatial Distribution of Objects
- ❖ The Signal of Noise Ratio

$$\mathcal{G} = \{F \circ C : F \in \mathcal{F}(L, \mathcal{W}, \mathcal{S}, \mathcal{B}), C \in \mathcal{C}(M, k, p, 1)\}$$

$$Y = F^* \circ C^*(X_{\text{obj}}),$$

References

Latif, G., Alghazo, J., Khan, M. A., Brahim, G. B., Fawagreh, K., & Mohammad, N. (2024). Deep convolutional neural network (CNN) model optimization techniques—Review for medical imaging. *AIMS Mathematics*, *9*(8), 20539-20571.

Li, Z., Liu, F., Yang, W., Peng, S., & Zhou, J. (2021). A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, *33*(12), 6999-7019.

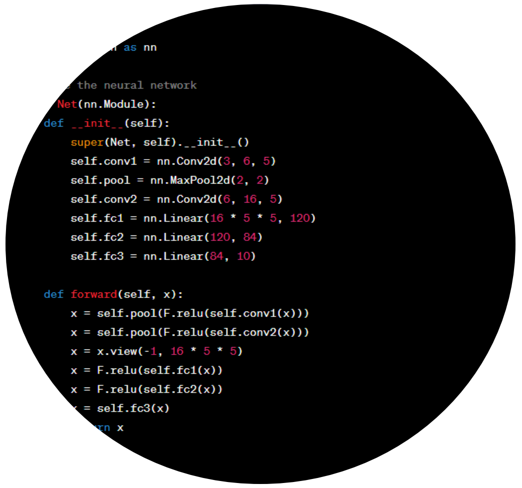
Kumar, Teerath, Rob Brennan, Alessandra Mileo, and Malika Bendechache. "Image data augmentation approaches: A comprehensive survey and future directions." *IEEE Access* (2024).

Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, *9*, 611-629.

Zhao, X., Wang, L., Zhang, Y., Han, X., Deveci, M., & Parmar, M. (2024). A review of convolutional neural networks in computer vision. *Artificial Intelligence Review*, *57*(4), 99.

Zou, Z., Chen, K., Shi, Z., Guo, Y., & Ye, J. (2023). Object detection in 20 years: A survey. *Proceedings of the IEEE*, *111*(3), 257-276.

How to succeed in this course?



```
     as nn

  the neural network
 Net(nn.Module):
def __init__(self):
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(3, 6, 5)
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, 5)
    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
      = self.fc3(x)
     rn x
```

Practice

Explore

Visualize

Discuss

Ask