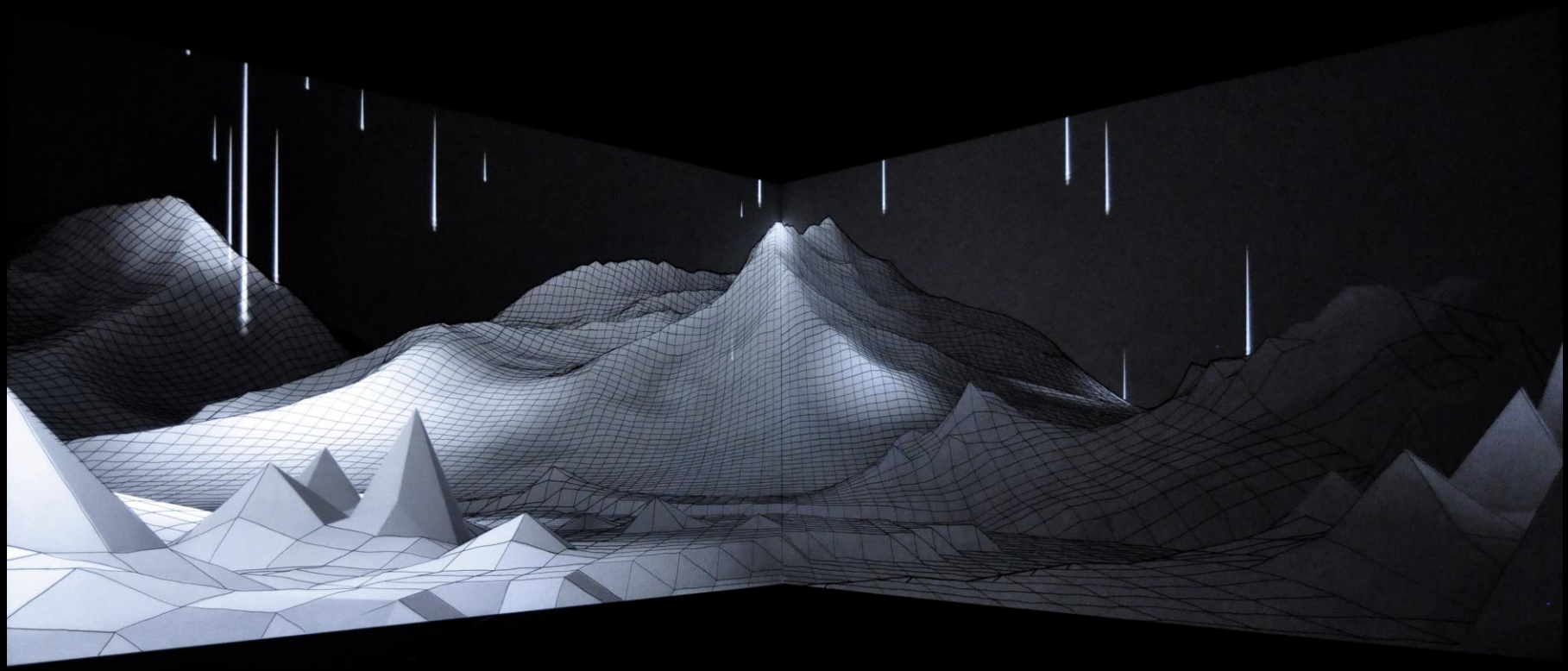
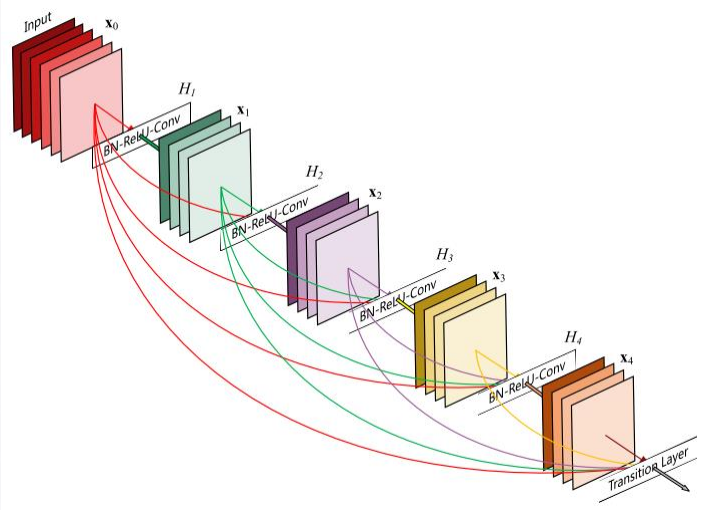


Convolutional Neural Networks
CS 410/510: CV & DL

Outline

- Convolutional Learning
- The Power of CNNs
- Structures of CNNs: Convolutional Layers, Strides, Padding, and Pooling
- Training CNNs
- Case Studies: Le-Net, AlexNet, VGG, ResNet, Inception, and Squeeze-Excitation
- Limitations of CNNs



Convolutional Learning

• Convolutional neural networks (CNNs) represent one of the early (and lasting) success stories of deep learning.

• CNNs are designed to work with grid-structured inputs, which have strong spatial dependences in local regions of the grid (e.g. 2D images).

• Features in CNNs **exploit this spatial dependencies**; as such, the vast majority applications of CNNs focus on image data. CNNs process tensor data (e.g., images with RGB channels, sets of images for video processing).

Convolutional Learning

• Convolutional neural networks (CNNs) represent one of the early (and lasting) success stories of deep learning.

• CNNs are designed to work with grid-structured inputs, which have strong spatial dependences in local regions of the grid (e.g. 2D images).

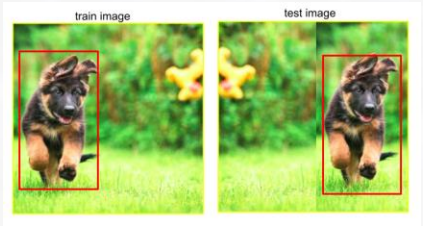
• Features in CNNs **exploit this spatial dependencies**; as such, the vast majority applications of CNNs focus on image data. CNNs process tensor data (e.g., images with RGB channels, sets of images for video processing).

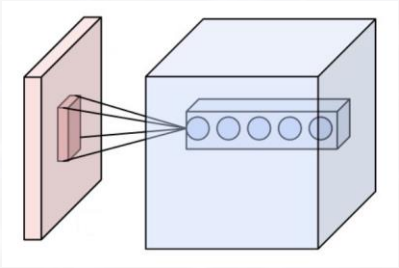
At their core, CNNs leverage several key design-related advantages:

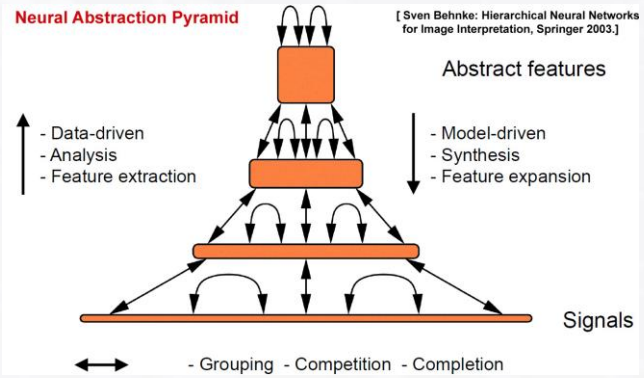
(1) The **compositional structure** of high-capacity deep networks for automated, hierarchical feature learning.

(2) **Parameter sharing** (for relatively compact model design)

(3) Learning of **translation invariance**

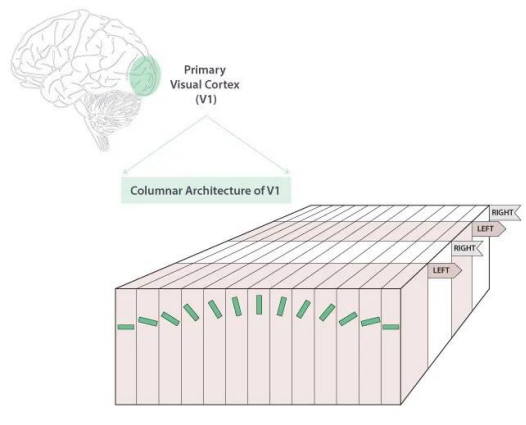






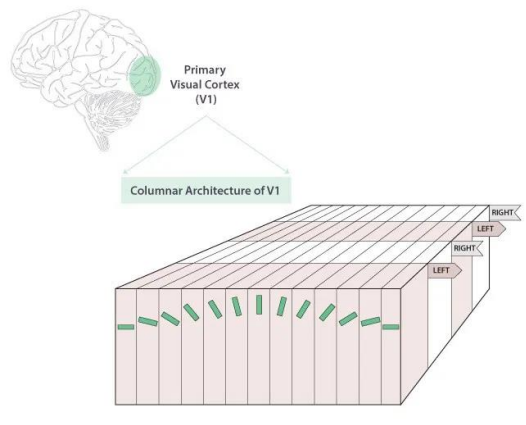
Convolutional Learning

• Early motivation for the development of CNNs was spurred in large part from the **Hubel and Wiesel** experiments, which demonstrated the existence of "specialized" cells sensitive to specific low-level visual features (e.g., edge orientation); moreover, these cells are arranged topographically so that nearby cells process information from nearby visual fields.

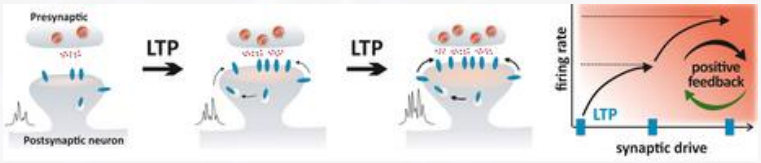


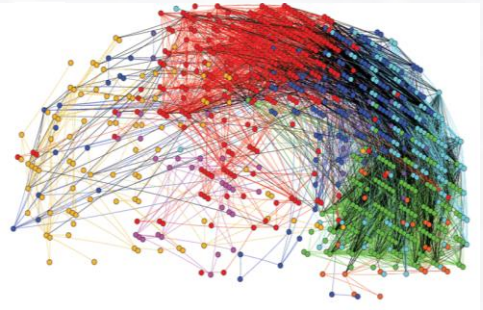
Convolutional Learning

• Early motivation for the development of CNNs was spurred in large part from the **Hubel and Wiesel** experiments, which demonstrated the existence of "specialized" cells sensitive to specific low-level visual features (e.g., edge orientation); moreover, these cells are arranged topographically so that nearby cells process information from nearby visual fields.



• In addition, the notion of "**cortical localization**" theory, attention and neural processing in relation to the **fovea**, research in **hierarchical brain functioning**, and the notion of **Hebbian plasticity** have all had a profound influence on the development of DL models, particularly CNNs in recent years.



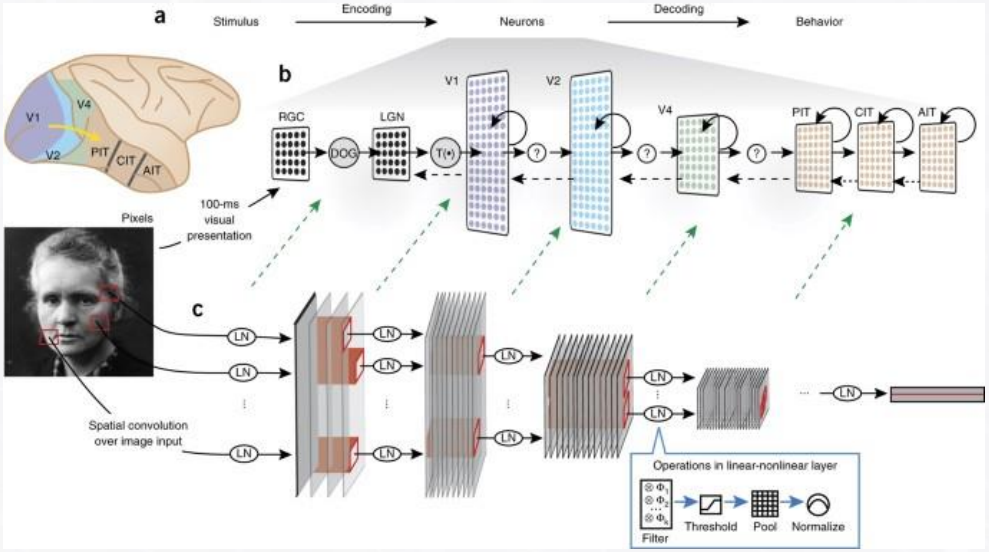


https://arxiv.org/abs/1004.3153

Convolutional Learning

• In biological brains, simple cells are connected in a layered architecture, and this discovery led to the conjecture that mammals use these different layers to construct portions of images at different levels of abstraction.

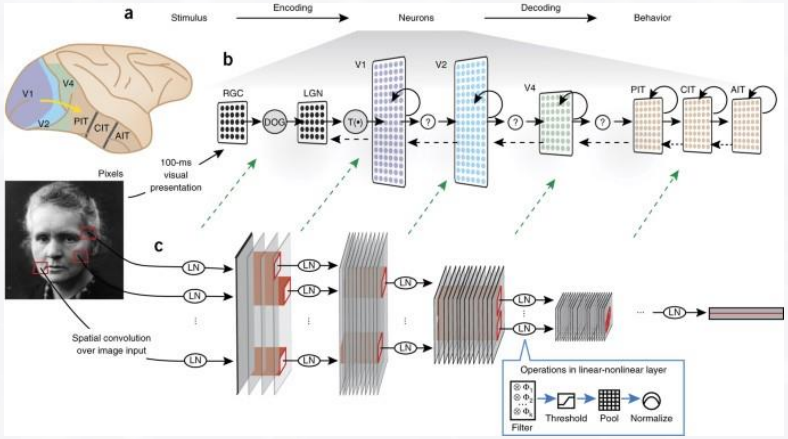
• From an ML perspective, this is akin to **hierarchical feature extraction**.

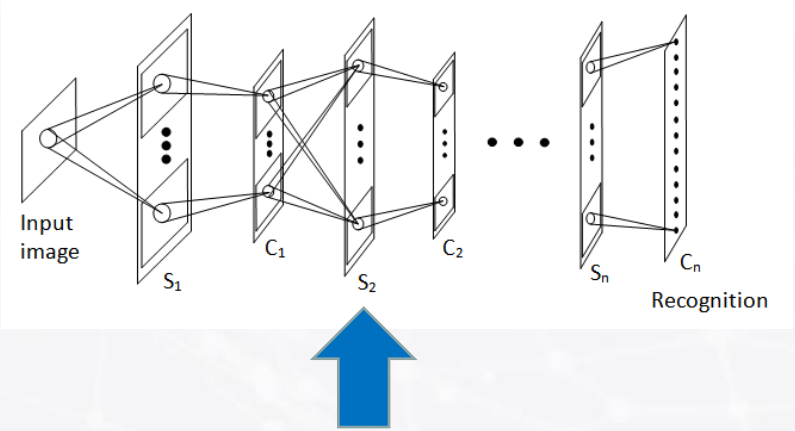


Convolutional Learning

• In biological brains, simple cells are connected in a layered architecture, and this discovery led to the conjecture that mammals use these different layers to construct portions of images at different levels of abstraction.

• From an ML perspective, this is akin to **hierarchical feature extraction**.





• One of the earliest precursors to the modern CNN was the **neocognitron** (Fukushima, 1980), which was directly inspired by the Hubel and Wiesel experiments.

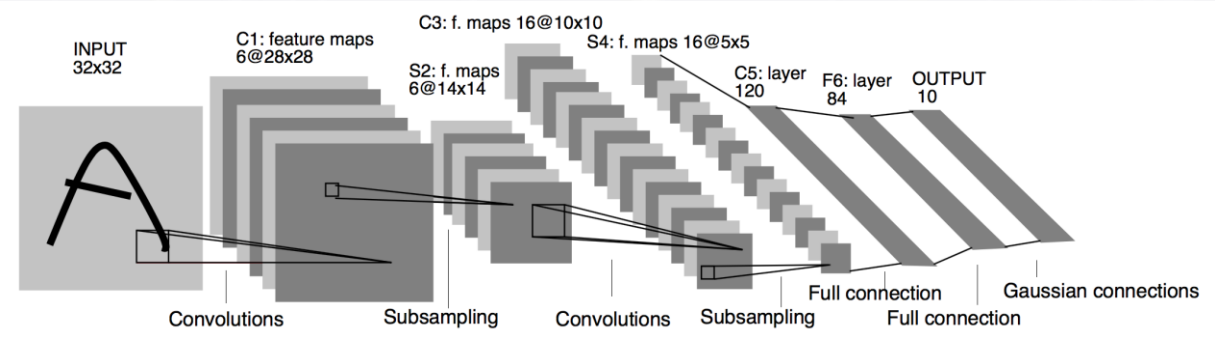
• The neocognitron consisted of two types of cells: **s-cells** (simple cells), learnable parameters for low-level feature extraction and **c-cells** (complex cells) that collate information from s-cells using a pooling operation. Notably, parameter sharing was not utilized; in addition, neocognitrons are typically trained using a self-organization workflow (not with backpropagation).

Convolutional Learning

• One of the seminal developments in the history of CNNs directly inspired by the neocognitron was the introduction of **LeNet** (1989) by Yann LeCun (2018 Turing Award).

• LeNet is a feed-forward CNN trained with backpropagation; many of the structural aspects (use of maxpooling, dense layers, backpropagation, etc.) of LeNet have since become canonized in the field.

• Notably, LeNet makes use of parameter sharing; LeCun demonstrated that minimizing the number of free parameters in a NN could improve its generalizability (a form of **regularization**).

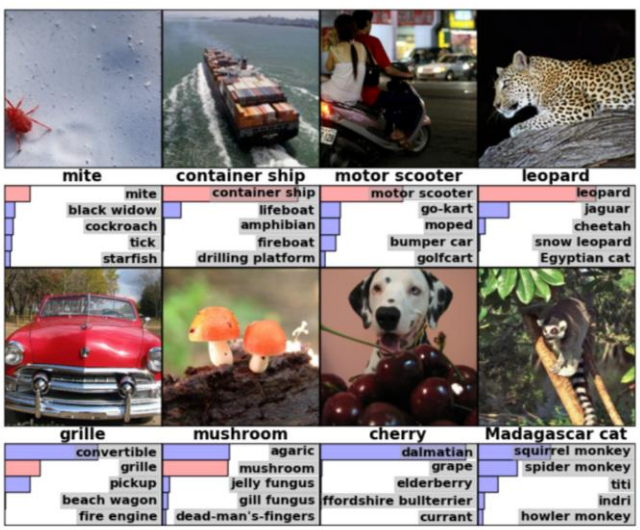


Convolutional Learning

• Even throughout much of the 1980s, a reliable solution to automated handwriting recognition was thought to be perhaps decades away!

• A prototype of LeNet was used by the US Postal Service to automate the reading of handwritten zip codes on letters. This is regarded as one of the first successful real-world applications of NNs in history.

The Power of CNNs



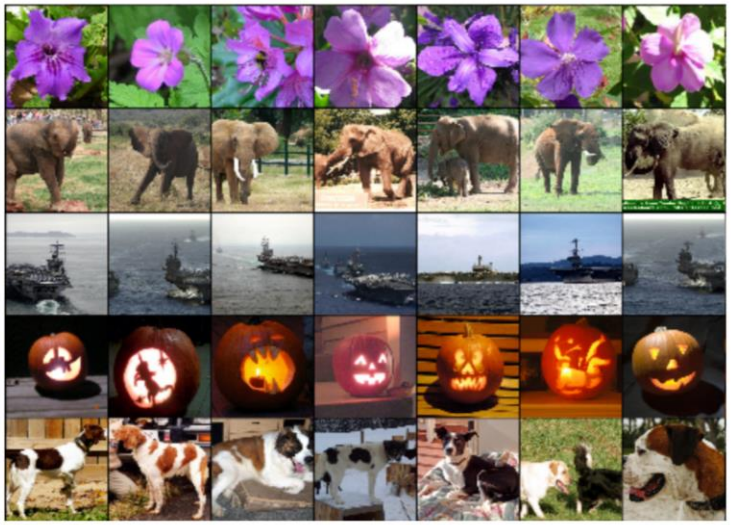
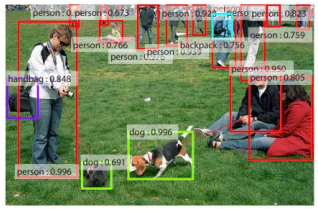
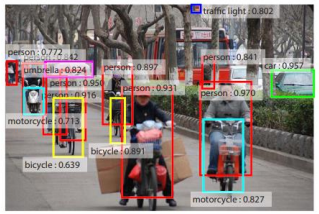
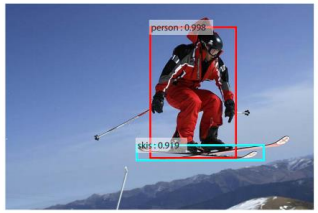


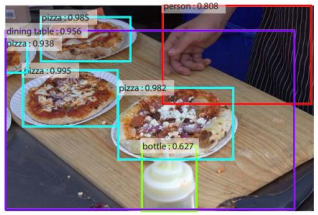
Image Classification

Image Retrieval





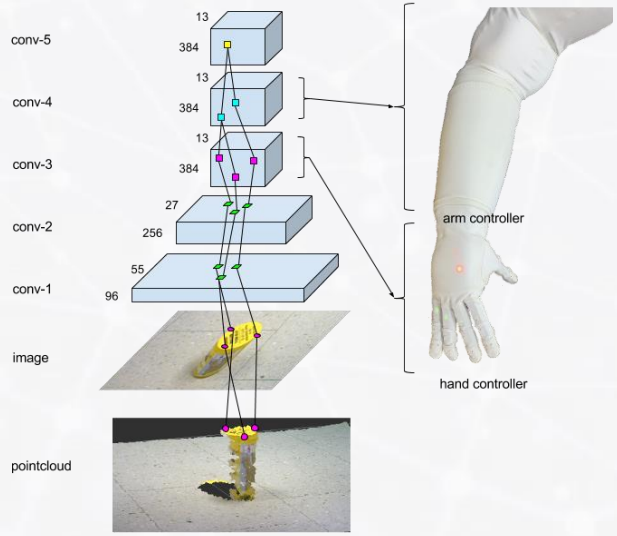




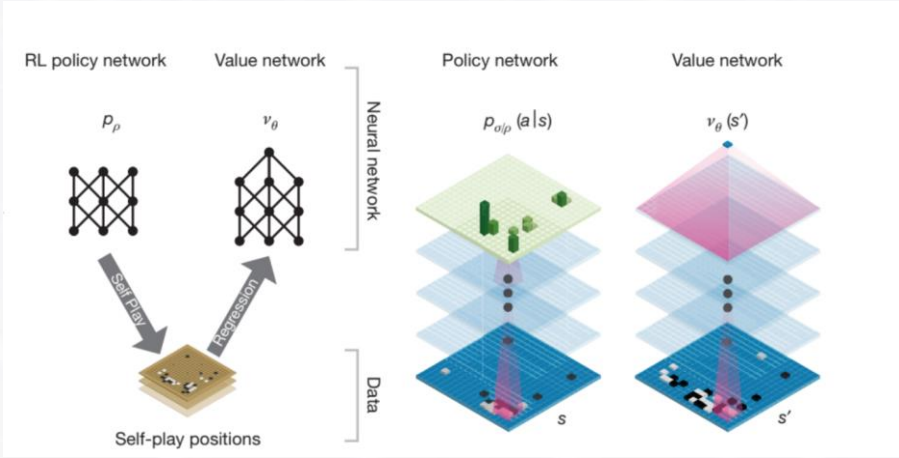
Object Localization

The Power of CNNs

Semantic Segmentation



Robot Control



Superhuman Game-Playing

The Power of CNNs

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

Object Captioning

thispersondoesnotexist.com

Synthetic Data Generation

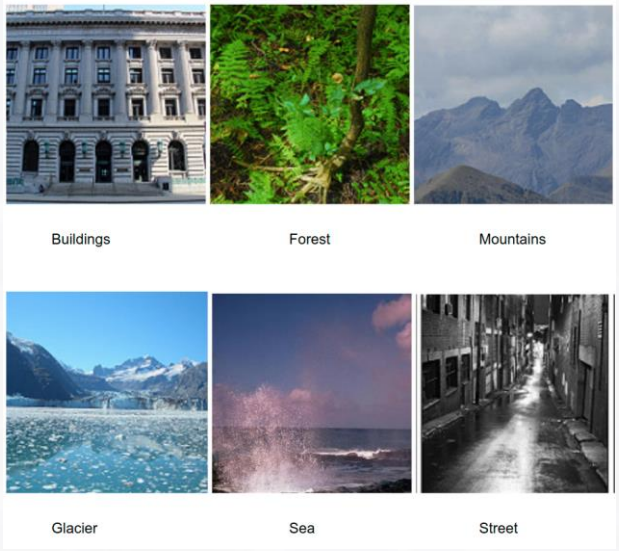
The Power of CNNs

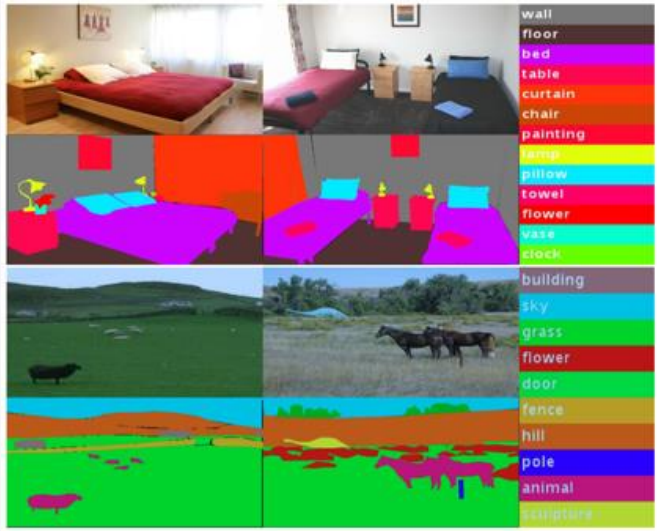
"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

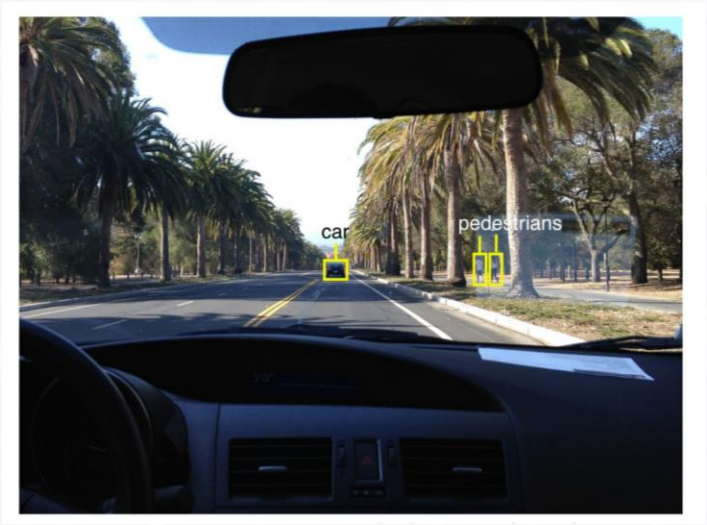
Object Captioning

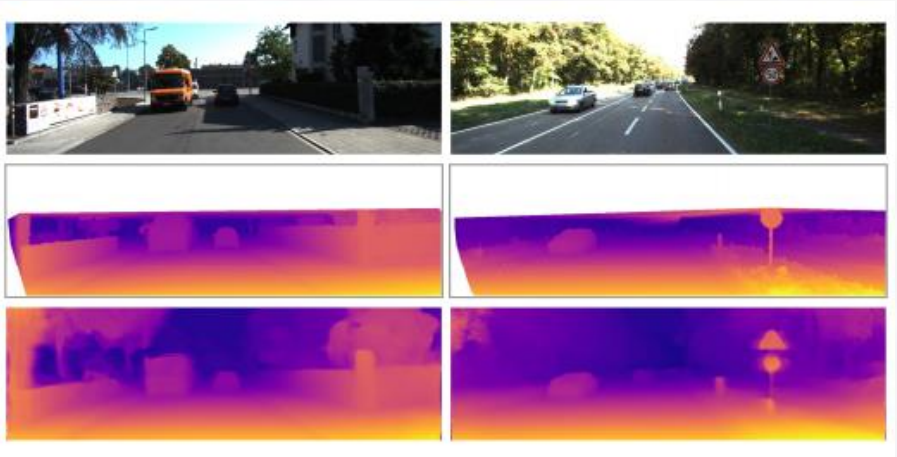




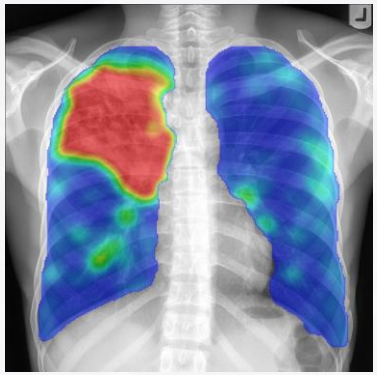
Scene Classification & Scene Understanding

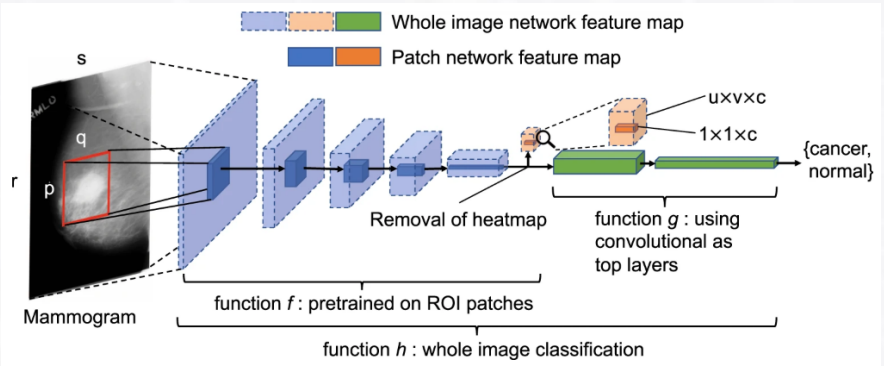
The Power of CNNs





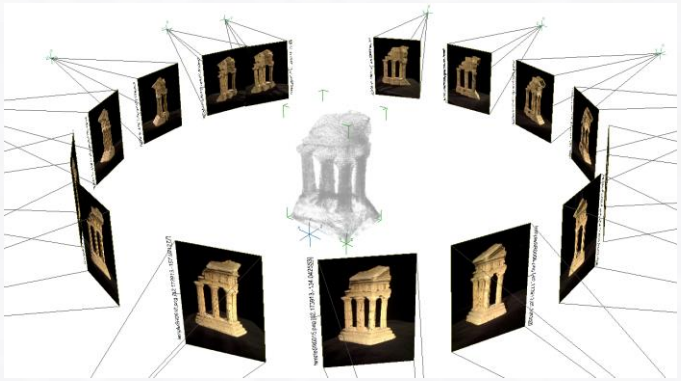
Self-Driving



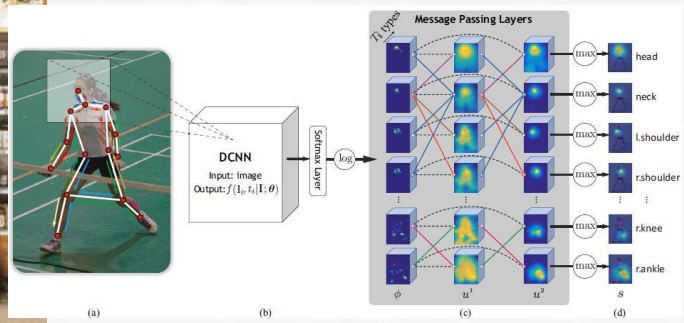


Medical Imaging Applications

The Power of CNNs



3D Scene Reconstruction & Scene Synthesis



Automation

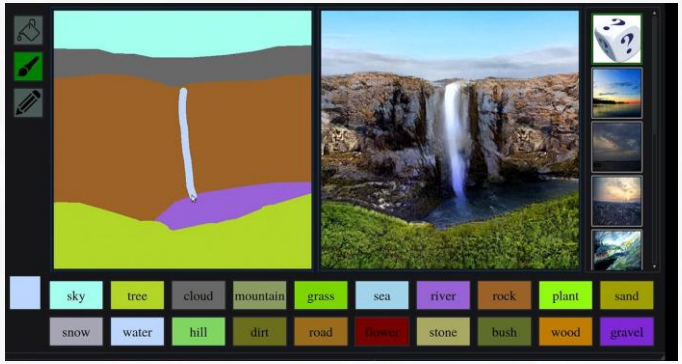
The Power of CNNs

Video Interpolation & Upsampling

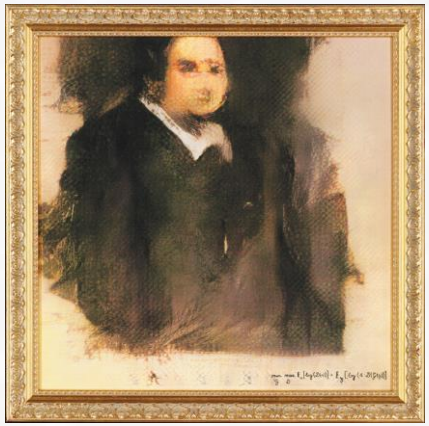
Video Game-Playing / Engineering

The Power of CNNs

Computational Creativity



http://nvidia-research-mingyuliu.com/gaugan



CV Datasets

•A factor that has played an important role in increasing the prominence of CNNs has been the annual **ImageNet competition** (ImageNet Large Scale Visual Recognition Challenge, ILSVRC)

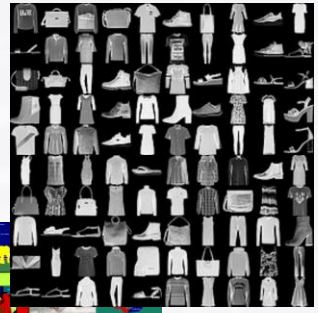
• **Fei-Fei Li** (Stanford) began working on the idea of ImageNet as early as 2006, when most in the field were focused on modeling and algorithm innovations. Labelling for ImageNet is primarily crowd sourced though mechanical turk.

• Today ImageNet (http://www.image-net.org/) contains ~14 million images in over 20k categories.

Other notable CV-related datasets:

• MNIST (benchmarking dataset; handwritten digits) https://deepai.org/dataset/mnist

• Fashion MNIST (benchmarking; 10 class grayscale images) https://github.com/zalandoresearch/fashion-mnist



• CIFAR-10, CIFAR-100 (benchmarking, classification) https://www.cs.toronto.edu/~kriz/cifar.html

• MS COCO (semantic segmentation, captions, keypoints) https://cocodataset.org/

• PASCAL VOC (classification) http://host.robots.ox.ac.uk/pascal/VOC/

• CelebA (facial features, GANs) http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

• Cityscapes dataset (automation, stereo, semantic segmentation) https://www.cityscapes-dataset.com/

• DAVIS (high resolution video segmentation) https://davischallenge.org/

*all datasets open-source

Structures of CNNs

• In CNNs, the states in each layer are **arranged according to a spatial grid structure**. These spatial relationships are inherited from one layer to the next; as each feature value is based on small spatial relationships among the grid cells.

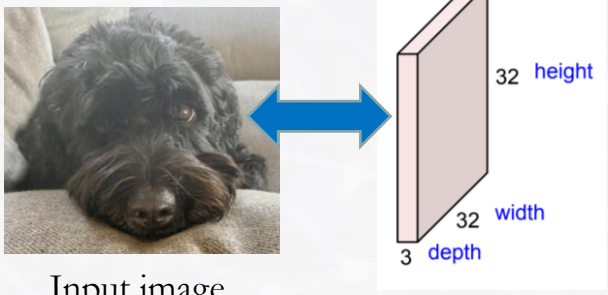
• Notice that spatial relationships are maintained layer by layer as the network processes input data (at least by conventional CNNs); this is a defining feature of CNNs and essential to most CV-related tasks.

Structures of CNNs

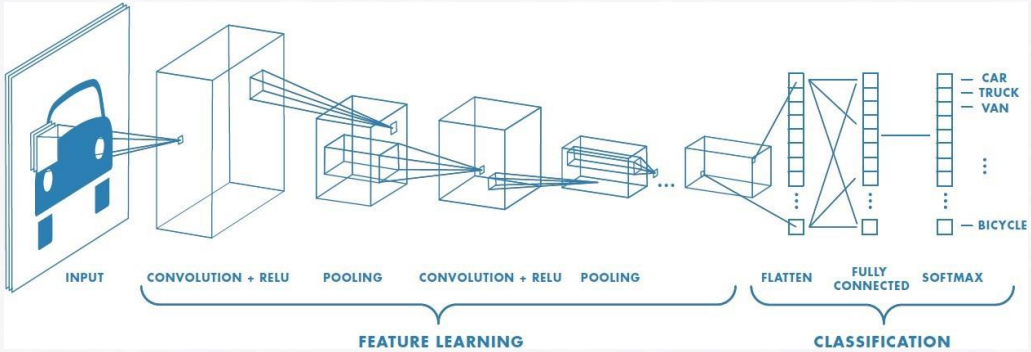
• In CNNs, the states in each layer are **arranged according to a spatial grid structure**. These spatial relationships are inherited from one layer to the next; as each feature value is based on small spatial relationships among the grid cells.

• Notice that <u>spatial relationships are maintained layer by layer</u> as the network processes input data (at least by conventional CNNs); this is a defining feature of CNNs and essential to most CV-related tasks.

• CNNs commonly consist of three types of layers: *convolution*, *pooling* (also: *pooling*), and *fully-connected* (also called *dense*) layers. **Each convolutional layer in a CNN is a 3D grid structure**, which has a height, width, and depth.

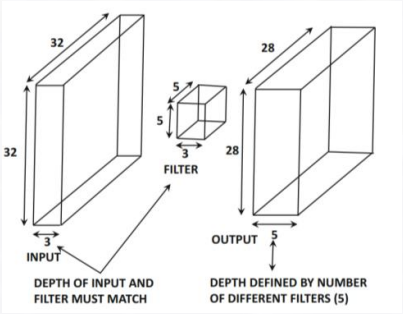


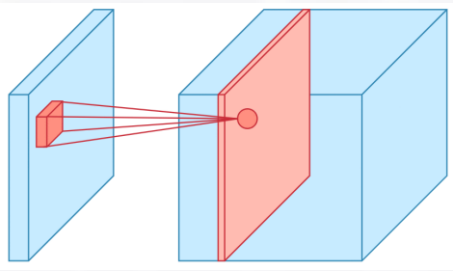
Input image



Structures of CNNs

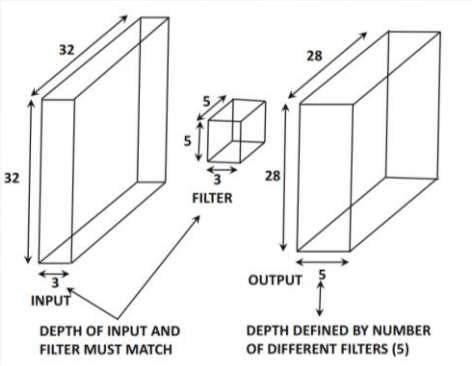
• The defining feature of a CNN is the **convolutional layer** wherein a sparse (due to parameter sharing) **convolution operation is performed between a patch of the layer input and a convolution filter**; this convolution operation is repeated for many patches over the entire layer input.

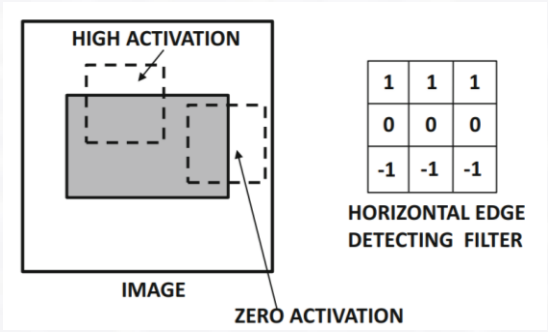




Structures of CNNs

• The defining feature of a CNN is the **convolutional layer** wherein a sparse (due to parameter sharing) **convolution operation is performed between a patch of the layer input and a convolution filter**; this convolution operation is repeated for many patches over the entire layer input.

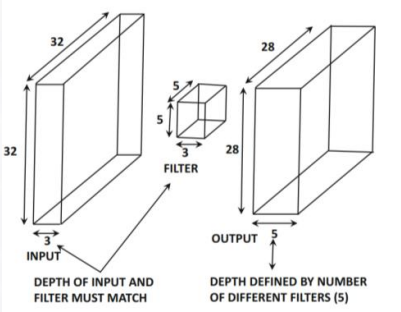


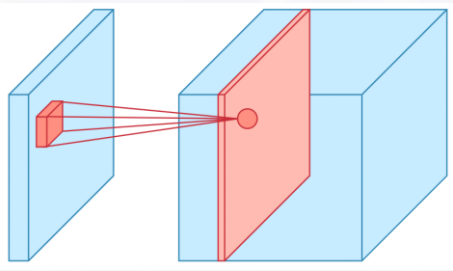


• The parameters of a CNN are organized into sets of **3D structural units** (*filters*, also: *kernels*). The filter is usually a small square (e.g. 3x3, 5x5); if the convolutional layer is of dimension 32x3x3, we say that the **depth** of the layer is 32 (i.e., 32 filters), where each filter is of size 3x3.

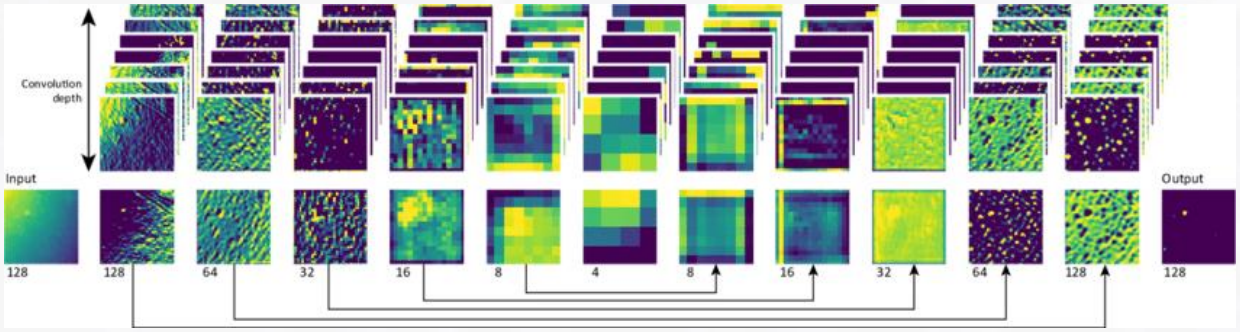
Structures of CNNs

• The defining feature of a CNN is the **convolutional layer** wherein a sparse (due to parameter sharing) **convolution operation is performed between a patch of the layer input and a convolution filter**; this convolution operation is repeated for many patches over the entire layer input.



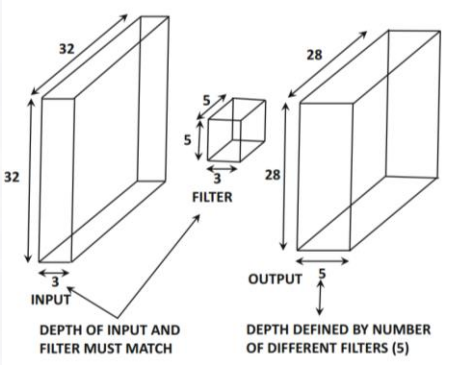


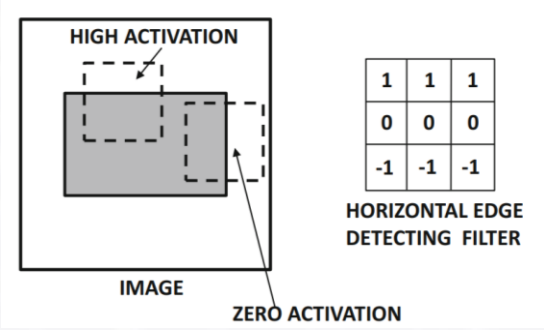
• The parameters of a CNN are organized into sets of **3D structural units** (*filters*, also: *kernels*). The filter is usually a small square (e.g., 3x3, 5x5); if the convolutional layer is of dimension 32x3x3, we say that the **depth** of the layer is 32 (i.e., 32 filters), where each filter is of size 3x3.



Structures of CNNs

• The defining feature of a CNN is the **convolutional layer** wherein a sparse (due to parameter sharing) **convolution operation is performed between a patch of the layer input and a convolution filter**; this convolution operation is repeated for many patches over the entire layer input.





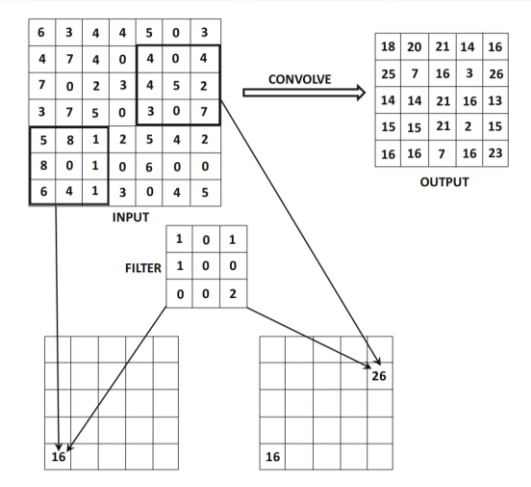
• The parameters of a CNN are organized into sets of **3D structural units** (*filters*, also: *kernels*). The filter is usually a small square (e.g. 3x3, 5x5); if the convolutional layer is of dimension 32x3x3, we say that the **depth** of the layer is 32 (i.e., 32 filters), where each filter is of size 3x3.

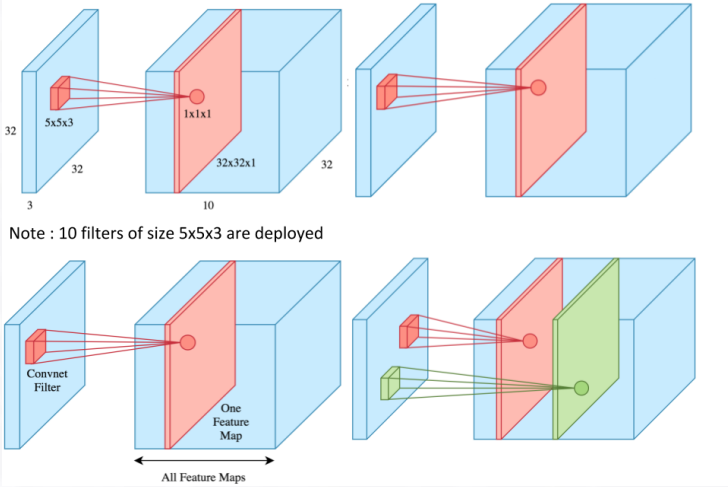
• As discussed previously, convolution is performed via a simple element-wise multiplication of a filter and input image patch. This multiplication yields an **activation** – a scalar value, indicating the degree of similarity between the patch and the filter – or put another way, the response of the simple cell (filter) to the stimulus. High activation indicates high similarity, etc.

• Notice that each filter is only locally-connected to a patch; this construct leads to parameter savings and furnishes CNNs with translational invariant properties.

Structures of CNNs

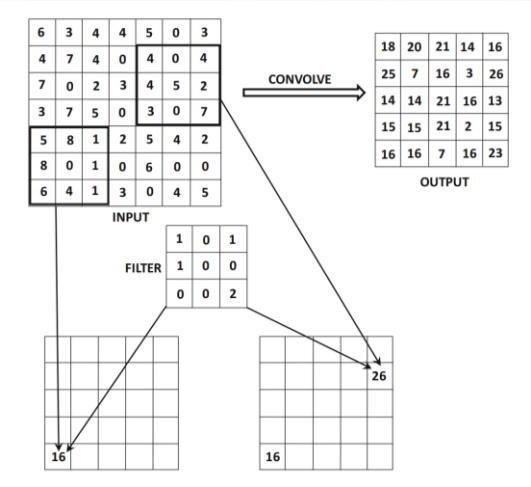
• **Convolution**: "slide the filter across the image", performing an element-wise multiplication at each spatial location. The step-size when we "slide" the filter is called the **stride**; if we slide one step at a time when convolving the filter and image, then stride = 1.

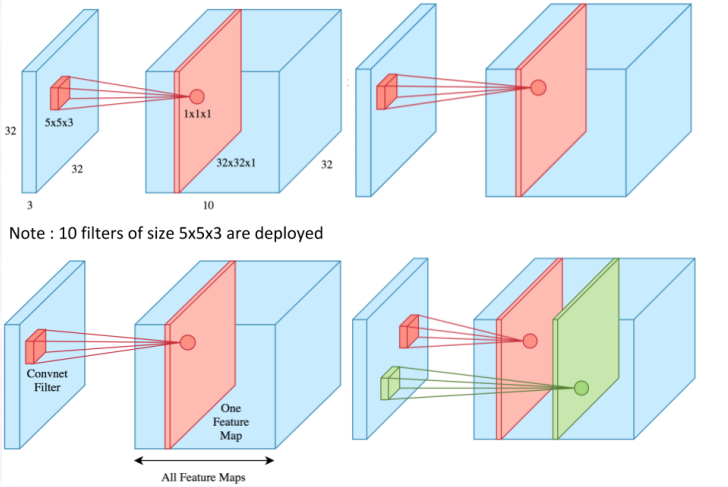




Structures of CNNs

• **Convolution**: "slide the filter across the image", performing an element-wise multiplication at each spatial location. The step-size when we "slide" the filter is called the **stride**; if we slide one step at a time when convolving the filter and image, then stride = 1.





• In the example above, the input is of dimension 7x7 and the filter is of dimension 3x3, and stride = 1. Notice that convolution results in a matrix of output activations of dimension 5x5.

• The following formula can be used to calculate the output (assumed square of dimension: $O \times O$) dimension when we convolve an input image ($I \times I$) with a filter of size ($K \times K$); typically, is K is odd:

$$O = \frac{I - K}{S} + 1$$

Confirming, we get: $\frac{7-3}{1} + 1 = 5$.

Structures of CNNs

•More formally, suppose that the ***p*th filter in the *q*th layer** of a CNN has parameters denoted by the 3D tensor $W^{(p,q)} = \left[W_{ijk}^{(p,q)}\right]$. The **feature maps in the *q*th layer** are represented by the 3D tensor $H^{(q)} = \left[h_{ijk}^{(q)}\right]$ (for instance $H^{(1)}$ represents the network input).

Structures of CNNs

•More formally, suppose that the pth filter in the qth layer of a CNN has parameters denoted by the 3D tensor $W^{(p,q)} = \left[W_{ijk}^{(p,q)}\right]$. The feature maps in the qth layer are represented by the 3D tensor $H^{(q)} = \left[h_{ijk}^{(q)}\right]$ (for instance $H^{(1)}$ represents the network input).

Then, the convolutional operations from the qth layer to the $(q+1)$th layer are defined as follows:

$$h_{ijp}^{(q+1)} = \sum_{r=1}^{K}\sum_{s=1}^{K}\sum_{k=1}^{K_d} w_{rsk}^{(p,q)} h_{i+r-1,\, j+s-1,k}^{(q)}$$

where the filter is of dimension $K \times K$, and the filter depth is given by K_d.

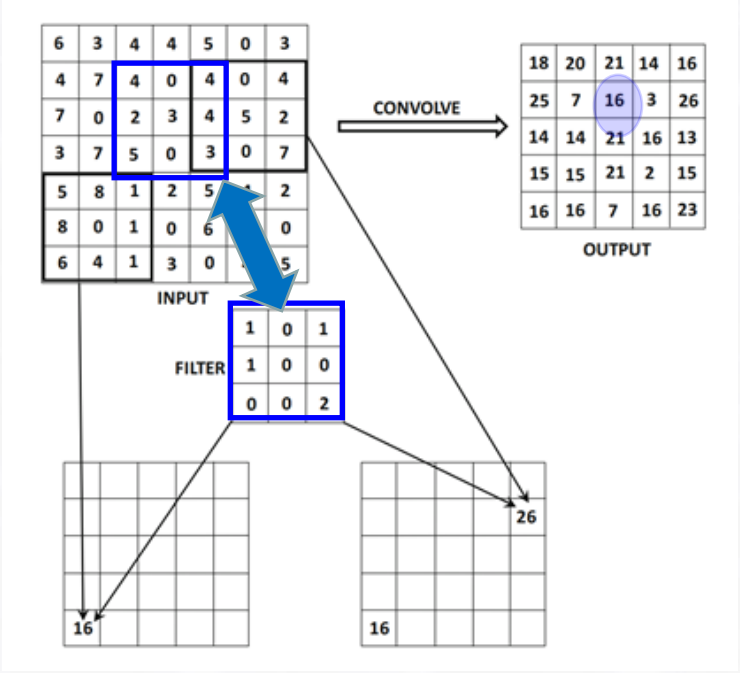
*Check that you understand this notation – it is simply a **mathematical formalization of convolution**, i.e., the summing of element-wise multiplication between a filter and feature map.

Structures of CNNs

•More formally, suppose that the pth filter in the qth layer of a CNN has parameters denoted by the 3D tensor $W^{(p,q)} = \left[W_{ijk}^{(p,q)}\right]$. The feature maps in the qth layer are represented by the 3D tensor $H^{(q)} = \left[h_{ijk}^{(q)}\right]$ (for instance $H^{(1)}$ represents the network input).

Then, the convolutional operations from the qth layer to the (q+1)th layer are defined as follows:

$$h_{ijp}^{(q+1)} = \sum_{r=1}^{K}\sum_{s=1}^{K}\sum_{k=1}^{K_d} w_{rsk}^{(p,q)} h_{i+r-1,\,j+s-1,k}^{(q)}$$



From the previous example:

$$h_{231}^{(q+1)} = w_{111}^{(p,q)} h_{2+1-1,3+1-1,1}^{(q)} + w_{121}^{(p,q)} h_{2+1-1,3+2-1,1}^{(q)} + w_{131}^{(p,q)} h_{2+1-1,3+3-1,1}^{(q)}$$
$$+ w_{211}^{(p,q)} h_{2+2-1,3+1-1,1}^{(q)} + w_{221}^{(p,q)} h_{2+2-1,3+2-1,1}^{(q)} + w_{231}^{(p,q)} h_{2+2-1,3+3-1,1}^{(q)}$$
$$+ w_{311}^{(p,q)} h_{2+3-1,3+1-1,1}^{(q)} + w_{221}^{(p,q)} h_{2+3-1,3+2-1,1}^{(q)} + w_{231}^{(p,q)} h_{2+3-1,3+3-1,1}^{(q)}$$
$$= 1\cdot h_{2,3,1}^{(q)} + 0\cdot h_{2,4,1}^{(q)} + 1\cdot h_{2,5,1}^{(q)}$$
$$+ 1\cdot h_{3,3,1}^{(q)} + 0\cdot h_{3,4,1}^{(q)} + 0\cdot h_{3,5,1}^{(q)}$$
$$+ 0\cdot h_{4,3,1}^{(q)} + 0\cdot h_{4,4,1}^{(q)} + 2\cdot h_{4,5,1}^{(q)}$$
$$h_{231}^{(q+1)} = 1\cdot 4 + 0\cdot 0 + 1\cdot 4 + 1\cdot 2 + 0\cdot 3 + 0\cdot 4 + 0\cdot 5 + 0\cdot 0 + 2\cdot 3 = 16$$

Structures of CNNs

Receptive Field of a Filter

• A convolution in the qth layer increases the receptive field of a feature from the qth layer to the $(q+1)$th layer. As such, each feature in the next layer captures a larger spatial region in the input layer.

• For instance, when using a 3×3 filter convolution successively in three layers, the activation in the first, second, and third hidden layers capture pixel regions of size 3×3, 5×5, and 7×7, respectively in the original input image.

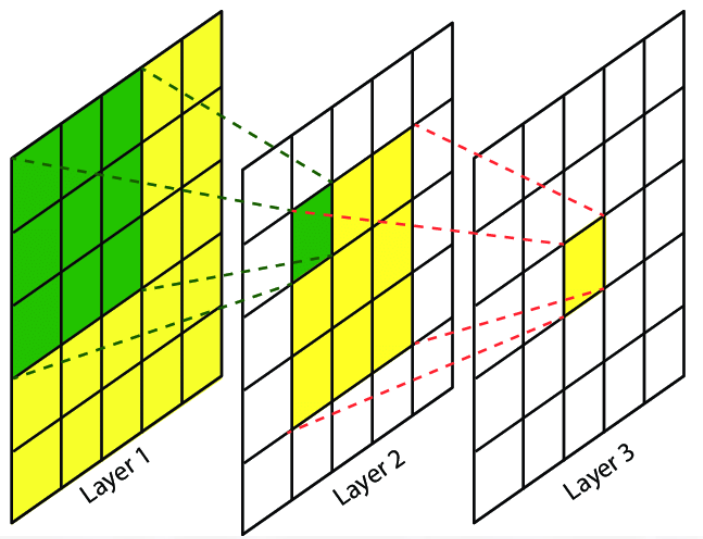
Structures of CNNs

Receptive Field of a Filter

• A convolution in the qth layer increases the receptive field of a feature from the qth layer to the $(q+1)$th layer. As such, each feature in the next layer captures a larger spatial region in the input layer.

• For instance, when using a 3×3 filter convolution successively in three layers, the activation in the first, second, and third hidden layers capture pixel regions of size 3×3, 5×5, and 7×7, respectively in the original input image.

• Recall that later layers in the CNN capture complex characteristics of the image over larger spatial regions. Other operations (e.g., **dilated convolution**) can also increase the receptive fields further.



Structures of CNNs

Padding

• In the previous examples, the convolution operation reduces the dimension of the feature map from one layer to the next (7×7 to 5×5); this is usually undesirable as we lose information along the periphery of the feature map.

• Conversely, we usually wish to **maintain the feature layer sizes** from one layer to the next when performing convolution. This is achieved simply by adding "padding" along the outside of an image/feature map.

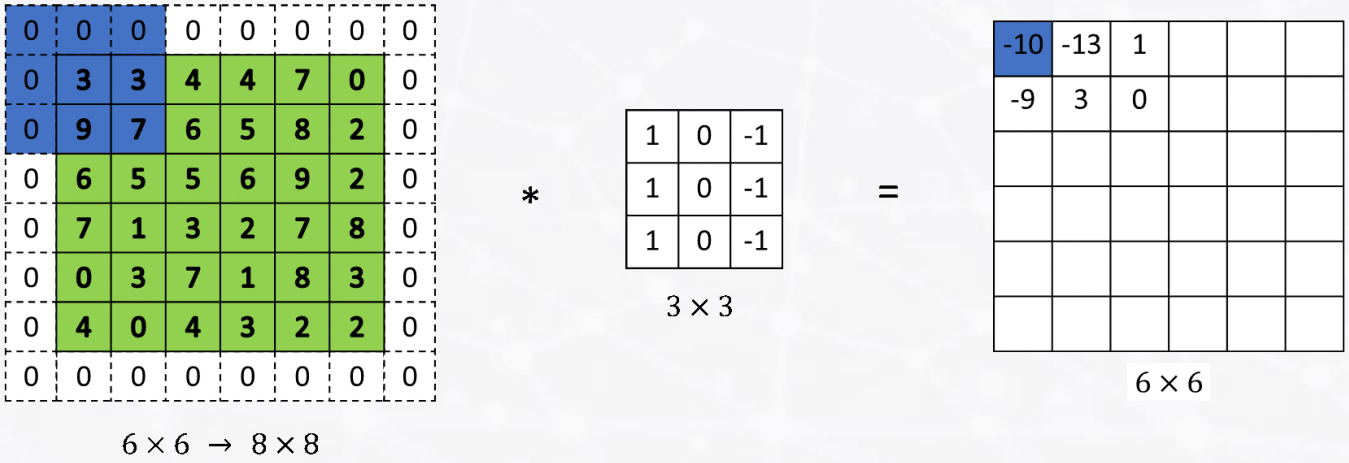
Structures of CNNs

Padding

• In the previous examples, the convolution operation reduces the dimension of the feature map from one layer to the next (7×7 to 5×5); this is usually undesirable as we lose information along the periphery of the feature map.

• Conversely, we usually wish to **maintain the feature layer sizes** from one layer to the next when performing convolution. This is achieved simply by adding "padding" along the outside of an image/feature map.

• In padding, one conventionally adds $\frac{K-1}{2}$ pixels all around the borders of the feature map in order to maintain the spatial footprint (e.g., if $K = 3$, padding $= \frac{3-1}{2} = 1$).



Structures of CNNs

Strides

• As mentioned, the **stride** is equivalent to the "step-size" as we slide the filter across the previous layer feature map. When $\text{stride} = 1$, for example, we "step" one pixel at a time.

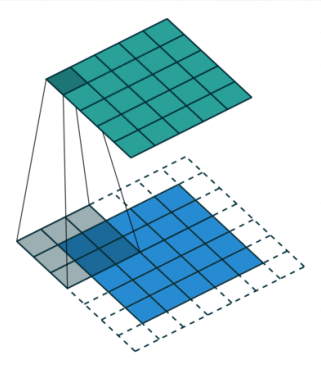
• As we increase the stride, we reduce the level of granularity of the convolution, which in turn can directly reduce the spatial footprint of the filter. Larger strides can be helpful in memory-constrained settings or to reduce overfitting.

Structures of CNNs

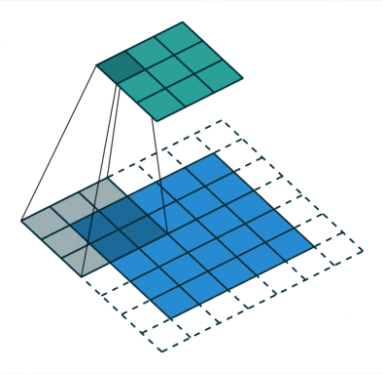
Strides

• As mentioned, the **stride** is equivalent to the "step-size" as we slide the filter across the previous layer feature map. When stride = 1, for example, we "step" one pixel at a time.

• As we increase the stride, we reduce the level of granularity of the convolution, which in turn can directly reduce the spatial footprint of the filter. Larger strides can be helpful in memory-constrained settings or to reduce overfitting.



stride=1, padding =1



stride=2, padding =1

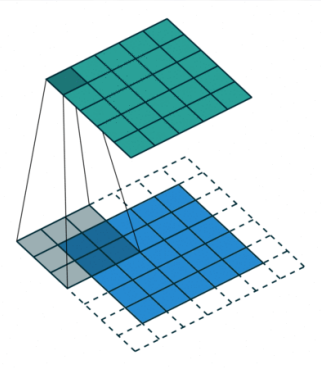
In summary: large strides increase the receptive field of each feature in the hidden layer, while reducing the spatial footprint (see animations).

Structures of CNNs

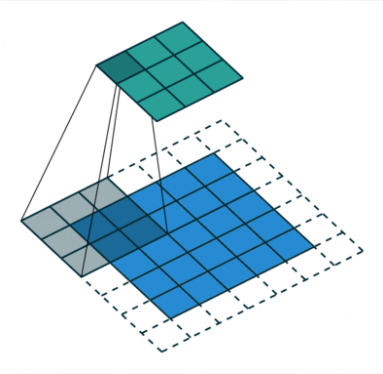
Strides

• As mentioned, the **stride** is equivalent to the "step-size" as we slide the filter across the previous layer feature map. When $\text{stride} = 1$, for example, we "step" one pixel at a time.

• As we increase the stride, we reduce the level of granularity of the convolution, which in turn can directly reduce the spatial footprint of the filter. Larger strides can be helpful in memory-constrained settings or to reduce overfitting.



stride=1, padding =1



stride=2, padding =1

Bias

• As with all NNs, one conventionally makes use of a **bias parameter** in each layer to improve the overall expressivity of the model. With CNNs, a bias is (conventionally) included for each filter. So, the pth filter in the qth layer will have a bias, denoted: $b^{(p,q)}$. The $(q+1)$th layer activation (including bias calculation) is given by:

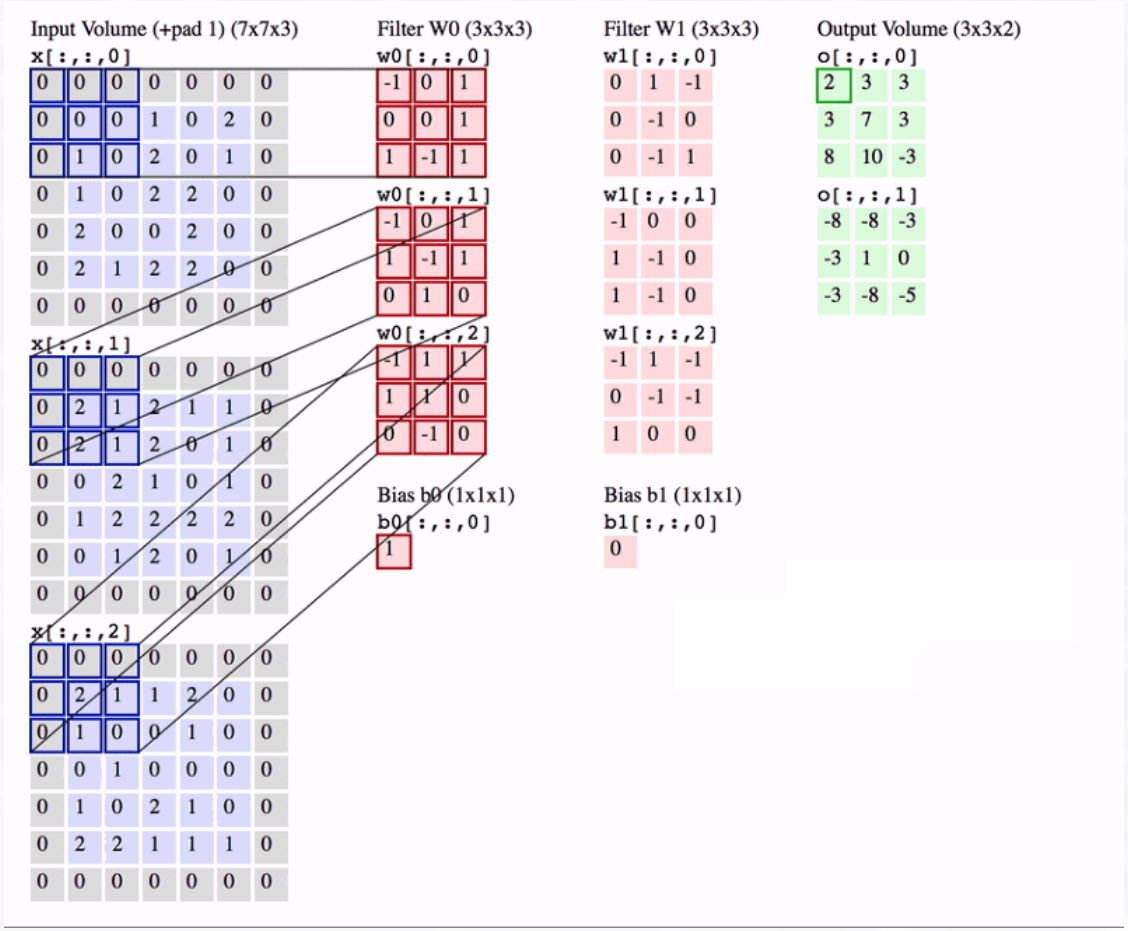
$$\mathbf{H}^{(q+1)} = \mathbf{W}^{(p,q)} \cdot \mathbf{H}^{(q)} + b^{(p,q)}$$

where the dot product above connotes a "matrix dot product" (i.e., elementwise product). As the bias is only a scalar value, the model biases generally present a small parameter overhead in relation to the entire network.

Structures of CNNs

• Here is an example using a set of 3D filters (one for each RGB channel), with biases included. In this case, each weight is convolved with an image patch – per channel. The sum of these features (the result of convolution) are added together, along with the bias to generate the output.

*Check to see that you can follow this example.



Structures of CNNs

Pooling

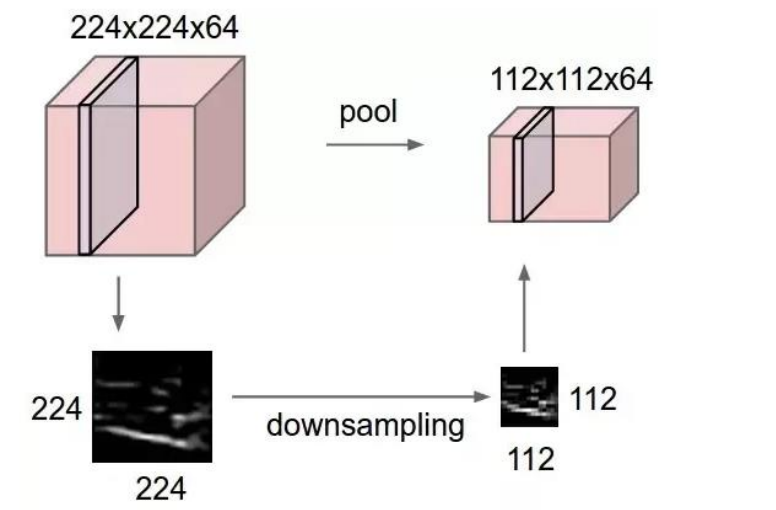
• The pooling operation reduces the dimension of the activation map in a CNN layer; oftentimes, pooling follows a sequence of several consecutive convolutional layers in a CNN. Pooling preserves the depth of the current network layer.

Structures of CNNs

Pooling

• The pooling operation reduces the dimension of the activation map in a CNN layer; oftentimes, pooling follows a sequence of several consecutive convolutional layers in a CNN. Pooling preserves the depth of the current network layer.

• Pooling represents a form of compute and memory overhead reduction – at the cost of losing useful structural information about the input image. *For this reason, pooling is today considered **somewhat controversial** amongst researchers. Pooling is akin to sampling an activation map, and often analogized as the *c*-cells in the visual cortex.



*See Hinton's talk on "What is wrong with CNNs": https://www.youtube.com/watch?v=Jv1VDdI4vy4

Structures of CNNs

Pooling

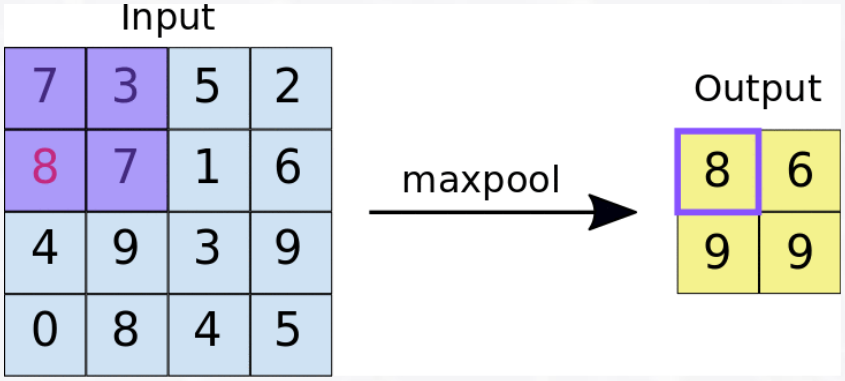
• Pooling is performed on an activation map; pooling operations have an associated **stride parameter** (default: stride = 1) and a **width parameter** (p_w). Pooling adds no additional tunable parameters to a model (stride and width are fixed for training/inference).

Structures of CNNs

Pooling

• Pooling is performed on an activation map; pooling operations have an associated **stride parameter** (default: stride = 1) and a **width parameter** (p_w). Pooling adds no additional tunable parameters to a model (stride and width are fixed for training/inference).

• The most common forms of pooling include (1) **maxpooling** and (2) **average pooling**. With maxpooling, we simply return the maximum activation within a sub-window of dimension $p_w \times p_w$; for average pooling, we return the average activation within the sub-window.



2×2 maxpooling, stride$= 2$

• Intuitively, pooling can be thought of as a means to propagate prominent visual features (i.e. high activations) through the network while saving memory/compute.

Structures of CNNs

Fully Connected (Dense) Layers

• Conventionally, the last layer(s) of a CNN consist of **fully-connected (FC) layers**. These layers function exactly as in a traditional feed-forward network. Commonly, the activation of the last convolutional layer in a CNN is "flattened"; this flattened vector serves as input to the first FC layer in

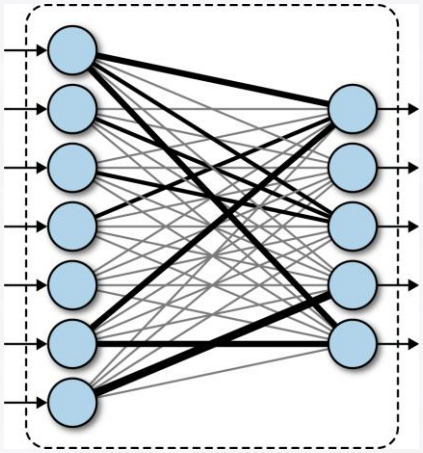
Structures of CNNs

Fully Connected (Dense) Layers

• Conventionally, the last layer(s) of a CNN consist of **fully-connected (FC) layers**. These layers function exactly as in a traditional feed-forward network. Commonly, the activation of the last convolutional layer in a CNN is "flattened"; this flattened vector serves as input to the first FC layer in the network.

• FC layers allow the network to collate all of the information provided by the sequence of convolutional and pooling layers found in the early layers of the network.

• Since the fully-connected layers are densely connected, the vast majority of parameters of a CNN lie in the FC layers. As with conventional NNs, following the FC layer(s) in a CNN, the output layer is designed in an application-specific way (i.e. for classification using softmax, segmentation, etc.).



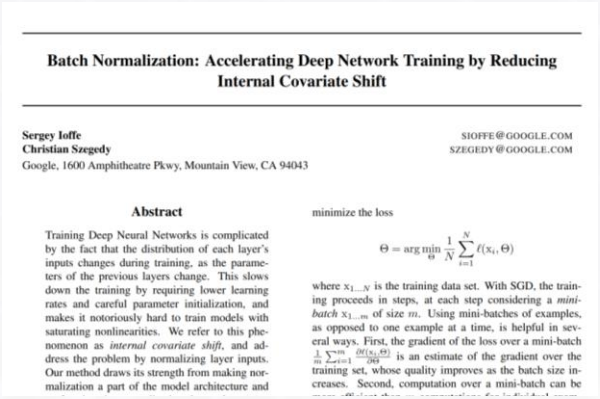
Batch Normalization

• **Batch normalization*** (BN) is a standard DL technique used to make training faster and more stable through normalization of the input layer by re-centering and re-scaling layer by layer.

Batch Normalization

• **Batch normalization*** (BN) is a standard DL technique used to make training faster and more stable through normalization of the input layer by re-centering and re-scaling layer by layer.

• It was initially claimed that BN reduces the instance of **internal covariate shift** (this claim has since been disputed), where parameter initialization and changes in the distribution of inputs of each layer affect the learning rate of the network. Some scholars conversely argue that BN smooths the objective function of the network.



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe IOFFE@GOOGLE.COM
Christian Szegedy SZEGEDY@GOOGLE.COM
Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043

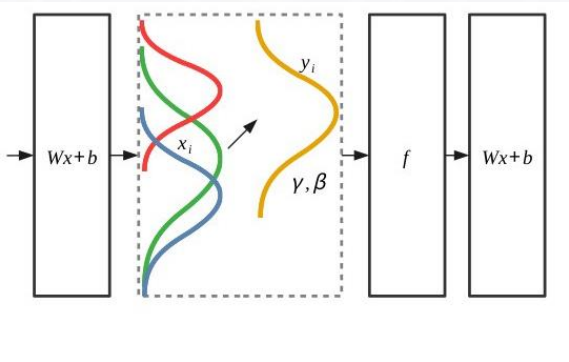
Abstract

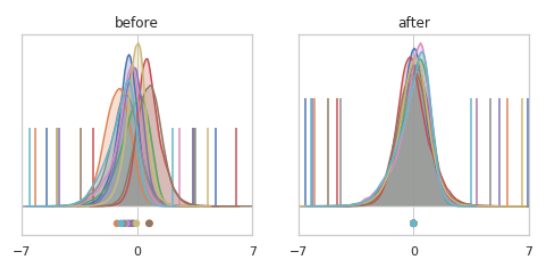
Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and

minimize the loss

$$\Theta = \arg\min_{\Theta} \frac{1}{N} \sum_{i=1}^{N} \ell(x_i, \Theta)$$

where $x_{1...N}$ is the training data set. With SGD, the training proceeds in steps, at each step considering a *mini-batch* $x_{1...m}$ of size m. Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch $\frac{1}{m}\sum_{i=1}^{m} \frac{\partial \ell(x_i, \Theta)}{\partial \Theta}$ is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a mini-batch can be





• Despite these debates as to the explicit DL phenomenon alleviated by BN, its effectiveness for network training is well established.

*https://arxiv.org/pdf/1502.03167.pdf

Batch Normalization

• Batch normalization fixes the means and variances of each layer's inputs. In tandem with stochastic optimization methods, we compute the mean and variance over mini-batches (denoted B) of size m.

$$\mu_B = \frac{1}{m}\sum_i x_i \qquad \frac{1}{m}\sum_i \left(x_i - \mu_B\right)^2$$

Batch Normalization

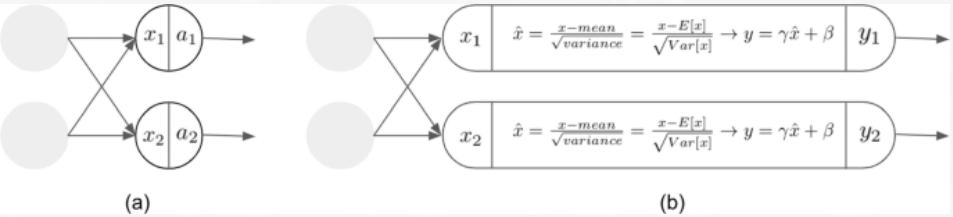
• Batch normalization fixes the means and variances of each layer's inputs. In tandem with stochastic optimization methods, we compute the mean and variance over mini-batches (denoted B) of size m.

$$\mu_B = \frac{1}{m}\sum_i x_i \qquad \frac{1}{m}\sum_i \left(x_i - \mu_B\right)^2$$

• For a layer (k) of the network with d-dimensional input, $x = \langle x^{(1)}, \ldots, x^{(d)} \rangle$ each dimension of its input is then normalized (i.e., re-centered and re-scaled) separately:

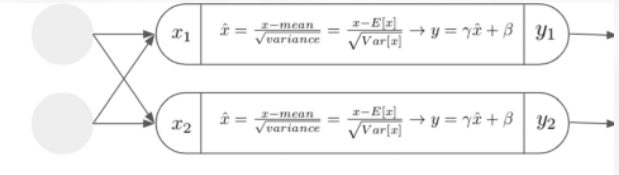
$$x_i^{(k)} \rightarrow \hat{x}_i^{(k)} = \frac{\hat{x}_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \varepsilon}}$$

• Where the transformation above shows the re-center and re-scale transformation; $\varepsilon > 0$ (fixed) is introduced for numerical stability (i.e., avoiding divide by zero).



Batch Normalization

$$x_i^{(k)} \rightarrow \hat{x}_i^{(k)} = \frac{\hat{x}_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \varepsilon}} \rightarrow y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

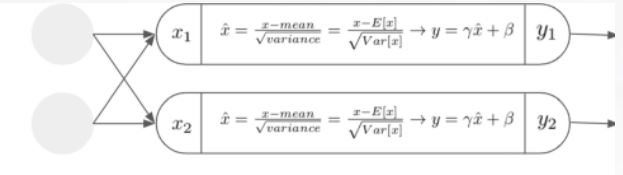


• After normalization, the layer inputs will have zero mean and unit variance. Finally, BN introduces two (new) trainable parameters per layer, a **scale parameter** γ and a **shift parameter** β.

• These parameters allow for the representation of the layer input (prior to normalization) to be restored – which is to say, we want the transformed distribution of the input data to be more homogeneous – but not necessarily centered at zero with unit variance (in the end).

Batch Normalization

$$x_i^{(k)} \rightarrow \hat{x}_i^{(k)} = \frac{\hat{x}_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \varepsilon}} \rightarrow y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$



• After normalization, the layer inputs will have zero mean and unit variance. Finally, BN introduces two (new) trainable parameters per layer, a **scale parameter** γ and a **shift parameter** β.

• These parameters allow for the representation of the layer input (prior to normalization) to be restored – which is to say, we want the transformed distribution of the input data to be more homogeneous – but not necessarily centered at zero with unit variance (in the end).

• The scale and shift parameters are learned through backpropagation. Note that BN is traditionally applied prior to non-linearity layers in a NN (although this convention is also debated amongst practitioners).

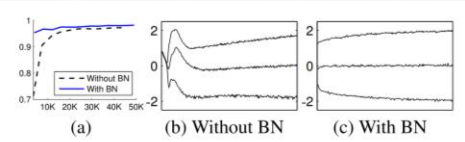


Figure 1. (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Batch Normalization

$$x_i^{(k)} \rightarrow \hat{x}_i^{(k)} = \frac{\hat{x}_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \varepsilon}} \rightarrow y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

```
def pure_batch_norm(X, gamma, beta, eps = 1e-5):
    if len(X.shape) not in (2, 4):
        raise ValueError('only supports dense or 2dconv')

    # dense
    if len(X.shape) == 2:
        # mini-batch mean
        mean = nd.mean(X, axis=0)
        # mini-batch variance
        variance = nd.mean((X - mean) ** 2, axis=0)
        # normalize
        X_hat = (X - mean) * 1.0 / nd.sqrt(variance + eps)
        # scale and shift
        out = gamma * X_hat + beta

    # 2d conv
    elif len(X.shape) == 4:
        # extract the dimensions
        N, C, H, W = X.shape
        # mini-batch mean
        mean = nd.mean(X, axis=(0, 2, 3))
        # mini-batch variance
        variance = nd.mean((X - mean.reshape((1, C, 1, 1))) ** 2, axis=(0, 2, 3))
        # normalize
        X_hat = (X - mean.reshape((1, C, 1, 1))) * 1.0 / nd.sqrt(variance.reshape((1, C, 1, 1)) +
        # scale and shift
        out = gamma.reshape((1, C, 1, 1)) * X_hat + beta.reshape((1, C, 1, 1))

    return out
```

Structures of CNNs

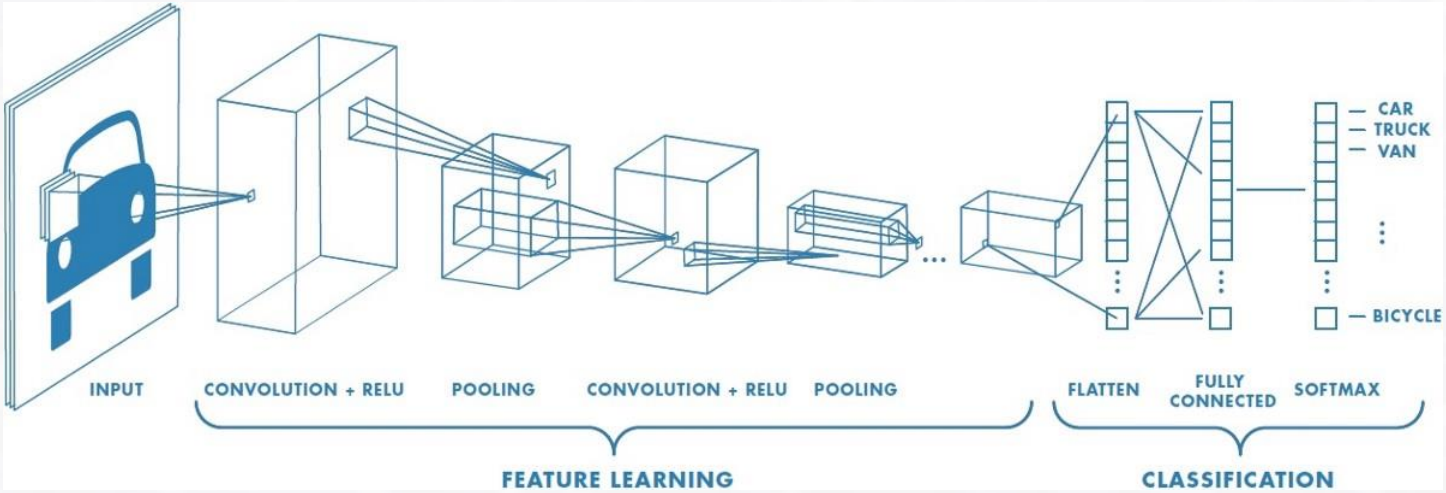
Putting it all together

• The convolution, pooling, and activation (RELU) layers are typically interleaved in a CNN in order to increase the expressive power of the network.

• RELU layers often follow the convolutional layers, just as a non-linear activation function typically follows the linear dot product in traditional NNs.

• After two or three sets of convolutional-RELU combinations, one might (in traditional CNN architectures) have a max-pooling layer, etc. Examples of this basic pattern are as follows:

CRCRP CRCRCRP CRCRPCRCRPCRCRPF



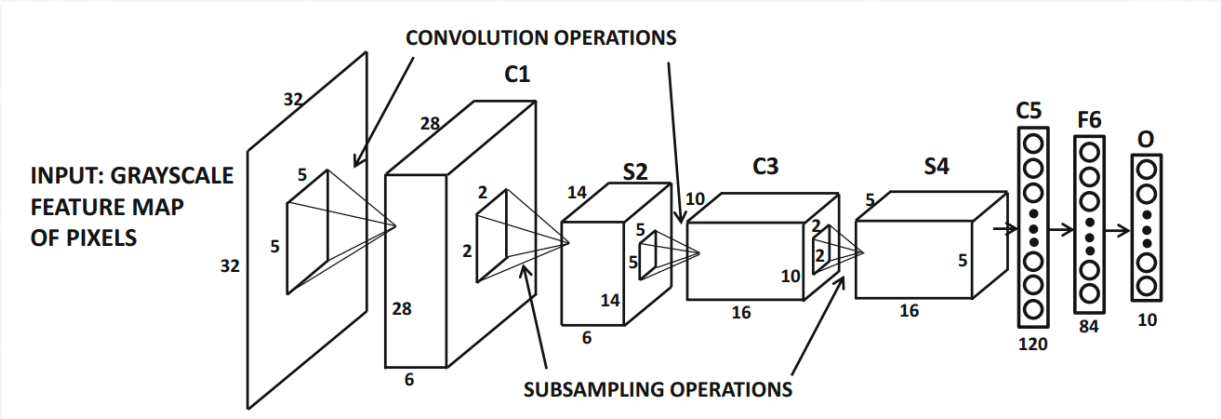
Structures of CNNs

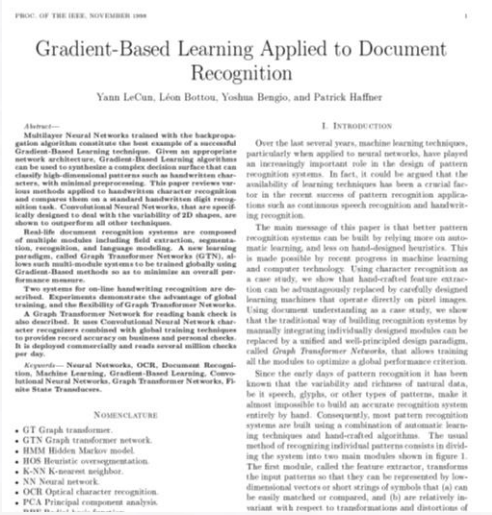
Le-Net-5 (LN5)

• Much of the preceding discussion regarding architectural conventions for CNNs was inspired by one of the earliest CNNs, **Le-Net-5*** (designed by Yann LeCun, Bengio). This simple architecture has had a profound influence on CNN designs following its introduction in 1998.

• The input to LN5 is a grayscale image; the network consists of two convolutional layers, two pooling layers and three fully-connected layers (as shown). Notice that 5x5 filters are used in the conv layers and 2x2 windows for "sub-sampling" (maxpooling); the FC layers have the structure 120-84-10 (number of neurons) – where 10 represents the number of output classes for classification (MNIST).

• Famously, LN5 was used commercially by banks to automating the reading of checks.





Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

*https://ieeexplore.ieee.org/document/726791

Structures of CNNs

CNN Parameter Counts

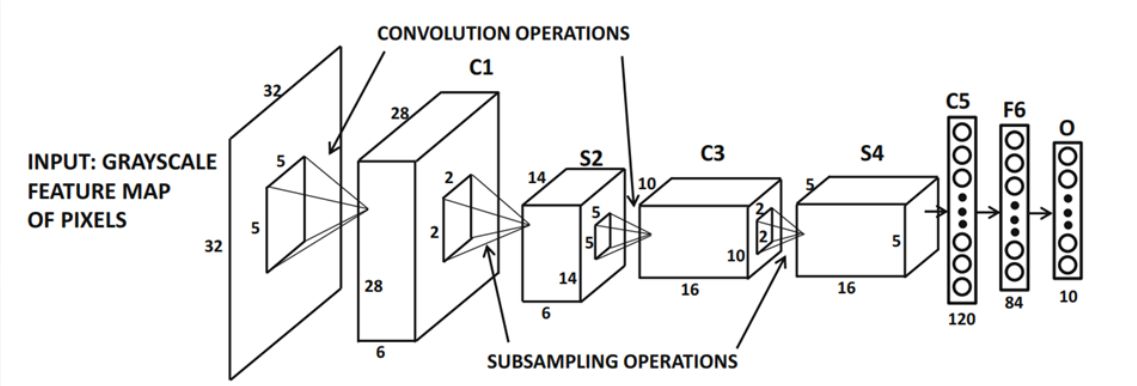
• **Input Layer**: No model parameter contribution.

• **Conv Layers**: Notice that technically, a conv layer comprises a 4D tensor of dimension (I,O,K,K), with I denoting the number of input channels, O the number of output channels, and the current layer filter is of dimension KxK. To conceptualize this 4D tensor, one can consider each kernel of dimension IxKxK with O such kernels.

In total, each conv layer therefore contains: $K \cdot K \cdot I \cdot O + O$ parameters (including a bias for each output kernel, indicated by the second term in the sum).

• **Pooling Layers**: No model parameter contribution.

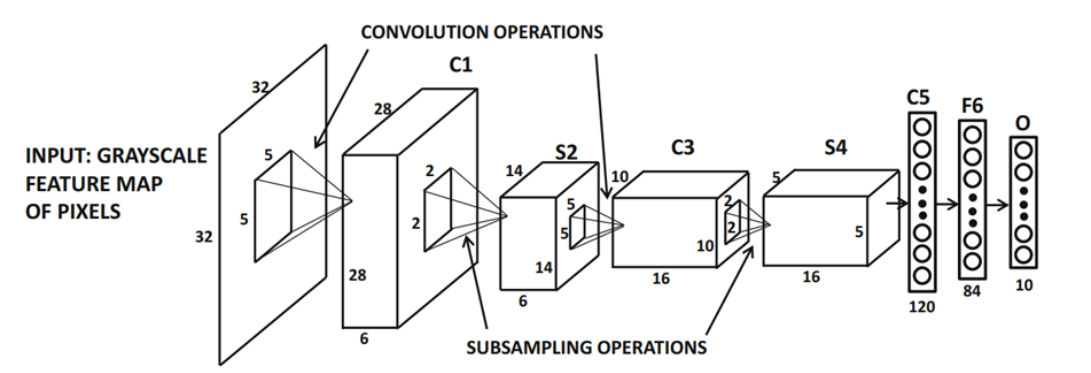
• **FC Layers**: Assume the input is of dimension I (neurons) and output of dimension O. Then, each FC layer contributes: $I \cdot O + O$ tunable parameters (including biases) to the CNN.



Structures of CNNs

CNN Parameter Counts

- **Input Layer**: None
- **Conv Layers**: $K \cdot K \cdot I \cdot O + O$
- **FC Layers**: $I \cdot O + O$




```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 6)         5·5·1·6+6=156
_________________________________________________________________
average_pooling2d_1 (Average (None, 12, 12, 6)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 16)          5·5·6·16+16=2416
_________________________________________________________________
average_pooling2d_2 (Average (None, 4, 4, 16)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 120)               (5·5·16)·120+120=48120
_________________________________________________________________
dense_2 (Dense)              (None, 84)                120·84+84=10164
_________________________________________________________________
dense_3 (Dense)              (None, 10)                84·10+10=850
=================================================================
Total Params: 61,706
```

TF model summary

Training CNNs

• The conventional process of training a CNN uses the backpropagation algorithm. There are primarily three types of layers: **convolutional**, **maxpooling** and **FC** layers.

• Notice that **FC layers** function as dense layers as in a standard FF, and therefore pose no additional difficulty for training. In other words, we perform backpropagation through an FC layer just as one would with a standard FF NN.

• For **maxpooling**, with no overlap between pools, one only needs to identify which unit is the maximum value in a pool. The partial derivative of the loss wrt the pooled state flows back to the unit with maximum value; all other entries in the grid are assigned a value of zero.

Training CNNs

• To calculate backpropagation through a convolutional layer, we apply a basic intuition: we observe that the forward pass through a convolutional layer amounts to a (sparse) **linear operation** (plus a bias)
– namely, we convolve the filter with different image patches.

• Using this insight, we can augment the convolution step as a **sparse matrix multiplication** -- in fact, this is how most modern SW libraries perform forward propagation for CNNs.

• As an added benefit, representing convolution operations in this way not only helps to simplify backpropagation for CNNs, but it additionally allows for improved efficiency in forward propagation (due to matrix operation optimization with GPUs, etc.).

Training CNNs

• Using this insight, we can augment the convolution step as a **sparse matrix multiplication** -- in fact, this is how most modern SW libraries perform forward propagation for CNNs. As an added benefit, representing convolution operations in this way not only helps to simplify backpropagation for CNNs, but it additionally allows for improved efficiency in forward propagation (due to matrix operation optimization with GPUs, etc.).

Training CNNs

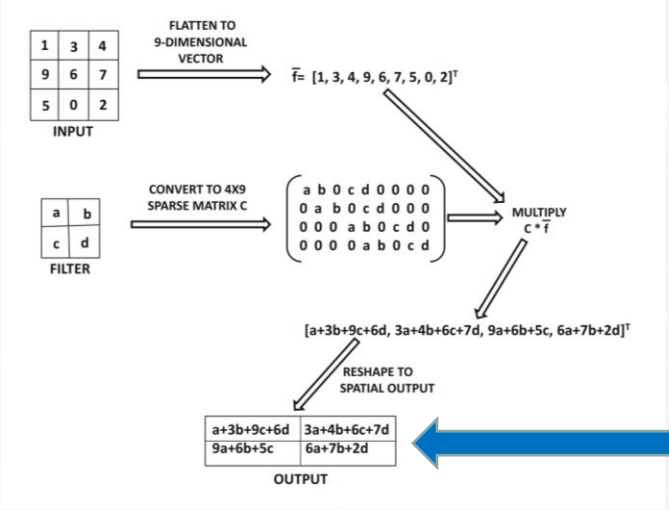
• Using this insight, we can augment the convolution step as a **sparse matrix multiplication** -- in fact, this is how most modern SW libraries perform forward propagation for CNNs. As an added benefit, representing convolution operations in this way not only helps to simplify backpropagation for CNNs, but it additionally allows for improved efficiency in forward propagation (due to matrix operation optimization with GPUs, etc.).

In more detail*:

(1) Convert the input, denoted f, into a flat (9D in the example) vector, this yields $\bar{f}$.

(2) Convert the filter (2x2 in the example) into a $(2 \cdot 2) \times 9$ **sparse matrix** C (notice that this matrix only contains tunable parameters per the filter of the CNN).

(3) To convolve the input f with the filter, perform the matrix multiplication: $C\bar{f}$ and reshape.



2x2 resultant matrix is equivalent to convolving the original input with filter

*This same process can be generalized to any filter and input.

Training CNNs

• With this general procedure to convert convolution operations into sparse matrix multiplication, backpropagation through a convolutional layer in a CNN is straightforward.

• To this end, suppose that $\bar{g}$ represents the flattened gradient vector of the loss function for our CNN (e.g., cross-entropy, MSE, etc.).

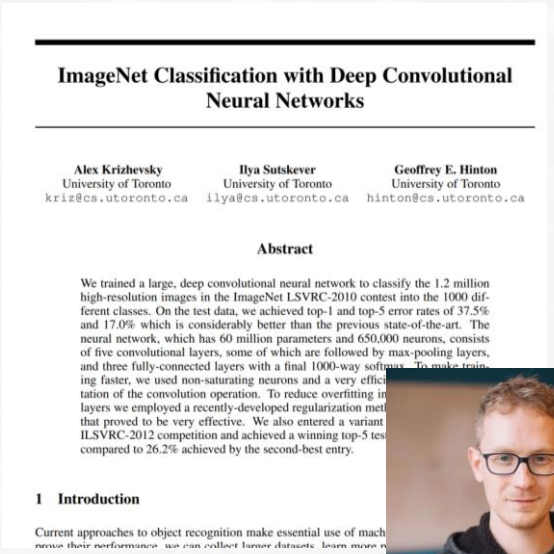
• One can show that backpropagation with respect to the convolutional layer represented by multiplication with the sparse matrix C (in the forward pass) is equivalent to multiplication by C^T. In short, to execute backpropagation through the convolutional layer, we simply compute: $C^T \bar{g}$.

• More generally, if the convolutional layer consists of K_d filters, the gradient with respect to the output maps produced by the convolutional layer is given by the sum: $\sum_{k=1}^{K_d} C_k^T \overline{g_k}$.

CNNs: Case Studies – AlexNet

AlexNet* (2012 ILSVRC winner)

• AlexNet represents perhaps the most influential CNN architecture in recent history; it embodies the first application of a truly "deep" CNN.



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky University of Toronto kriz@cs.utoronto.ca

Ilya Sutskever University of Toronto ilya@cs.utoronto.ca

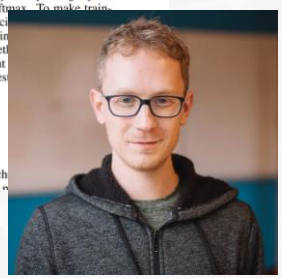
Geoffrey E. Hinton University of Toronto hinton@cs.utoronto.ca

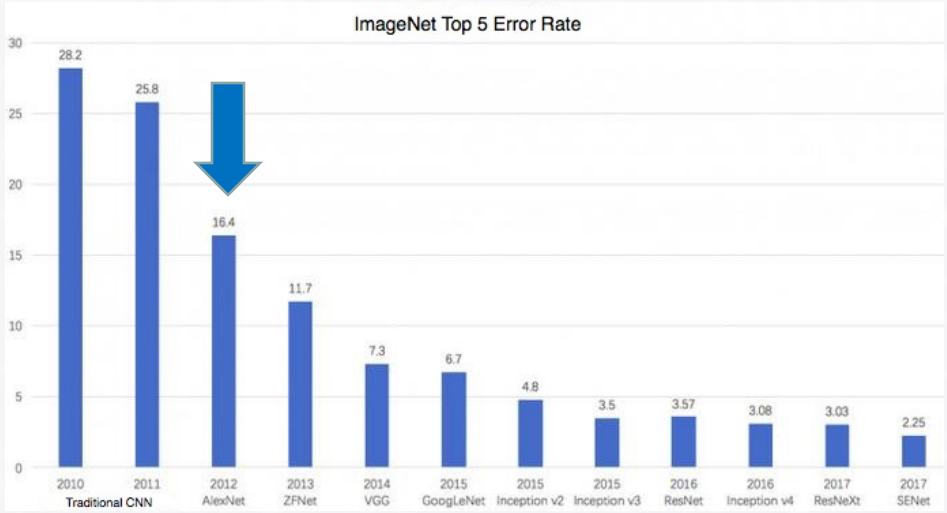
Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

1 Introduction

Current approaches to object recognition make essential use of machine learning methods. To improve their performance, we can collect larger datasets, learn more p

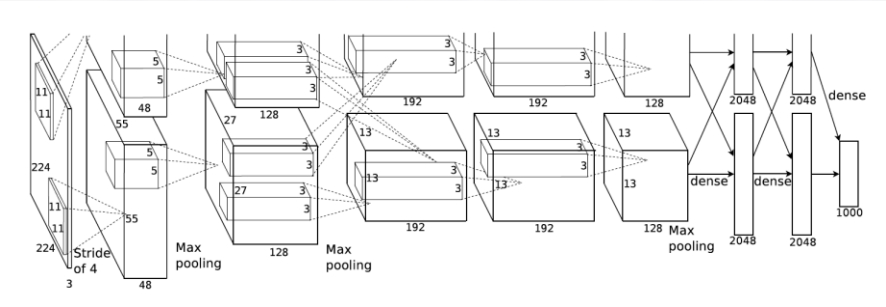




*https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

CNNs: Case Studies – AlexNet

AlexNet (2012 ILSVRC winner)



AlexNet Network - Structural Details												
Input			Output			Layer	Stride	Pad	Kernel size		in	out	# of Param
227	227	3	55	55	96	conv1	4	0	11	11	3	96	34944
55	55	96	27	27	96	maxpool1	2	0	3	3	96	96	0
27	27	96	27	27	256	conv2	1	2	5	5	96	256	614656
27	27	256	13	13	256	maxpool2	2	0	3	3	256	256	0
13	13	256	13	13	384	conv3	1	1	3	3	256	384	885120
13	13	384	13	13	384	conv4	1	1	3	3	384	384	1327488
13	13	384	13	13	256	conv5	1	1	3	3	384	256	884992
13	13	256	6	6	256	maxpool5	2	0	3	3	256	256	0
						fc6			1	1	9216	4096	37752832
						fc7			1	1	4096	4096	16781312
						fc8			1	1	4096	1000	4097000
						Total							**62,378,344**

• Some architectural details: input (ImageNet) is 227x227x3, uses 96 filters of size 11x11x3 (considered a large filter size by today's standards) with stride=4, followed by maxpooling (with overlapping).

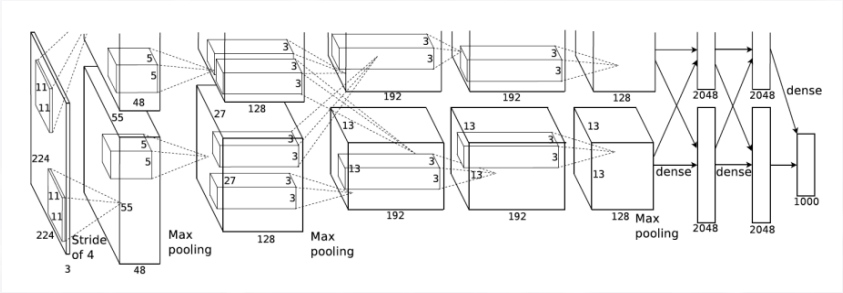
• This pattern of **CRPCRP** is followed in the subsequent layers; the next block structure contains three consecutive convolutional layers followed by pooling.

• The back-end of the network consists of three consecutive FC layers; notice that **94%**(!) of the model parameters are contained in the FC layers.

*Observe the structural similarities *cf.* Le-Net-5.

CNNs: Case Studies – AlexNet

AlexNet (2012 ILSVRC winner)



AlexNet Network - Structural Details												
Input			Output			Layer	Stride	Pad	Kernel size		in	out	# of Param
227	227	3	55	55	96	conv1	4	0	11	11	3	96	34944
55	55	96	27	27	96	maxpool1	2	0	3	3	96	96	0
27	27	96	27	27	256	conv2	1	2	5	5	96	256	614656
27	27	256	13	13	256	maxpool2	2	0	3	3	256	256	0
13	13	256	13	13	384	conv3	1	1	3	3	256	384	885120
13	13	384	13	13	384	conv4	1	1	3	3	384	384	1327488
13	13	384	13	13	256	conv5	1	1	3	3	384	256	884992
13	13	256	6	6	256	maxpool5	2	0	3	3	256	256	0
						fc6			1	1	9216	4096	37752832
						fc7			1	1	4096	4096	16781312
						fc8			1	1	4096	1000	4097000
						Total							**62,378,344**

Notable innovations of AlexNet

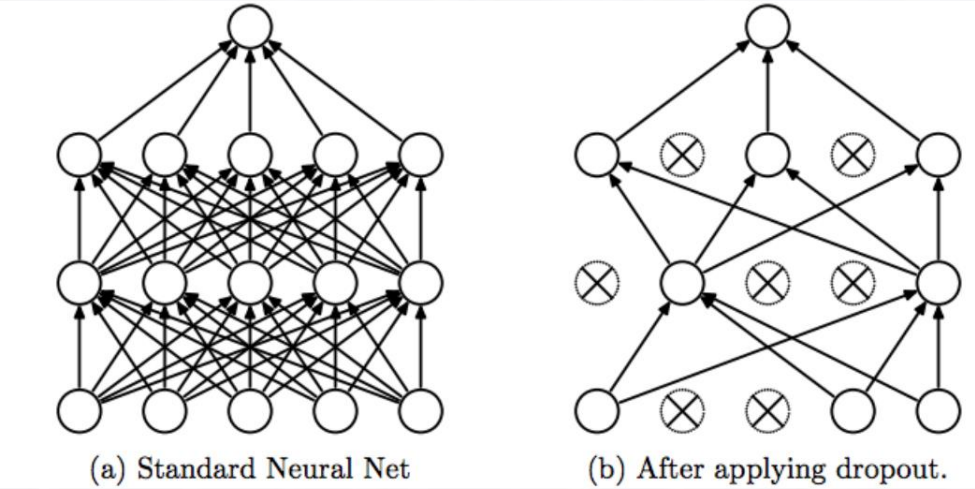
• Sheer size (and depth): **~62M parameters**; previous year SOTA top-5 error rate reduced by 35%.

• Use of **RELU** (in place of tanh which was standard at the time); RELU activation subsequently became *de facto* activation for DL models.

• Use of GPUs to accelerate training. In fact, the network was trained on GTX 580 GPU with 3GB of RAM, which could not fit the entire network and so it was partitioned across two GPUs (see image).

• Introduction of **Dropout** regularization method.

CNNs: Case Studies – Dropout



• Dropout (Srivastava et al., 2014) provides a computationally inexpensive but powerful method of regularizing a broad family of models (it is akin to *bagging*).

• Dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from an underlying base network. With bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i. Dropout aims to approximate this process, but with an exponentially large number of NNs.

• In practice, each time we load an example into a minibatch for training, we randomly sample a different binary mask to apply to all input and hidden units in the network; the mask is sampled independently for each unit (e.g. 0.8 probability for including an input unit and 0.5 for hidden units).

• In the case of bagging, the models are all independent; for dropout, the models share parameters.

CNNs: Case Studies – AlexNet

AlexNet (2012 ILSVRC winner)

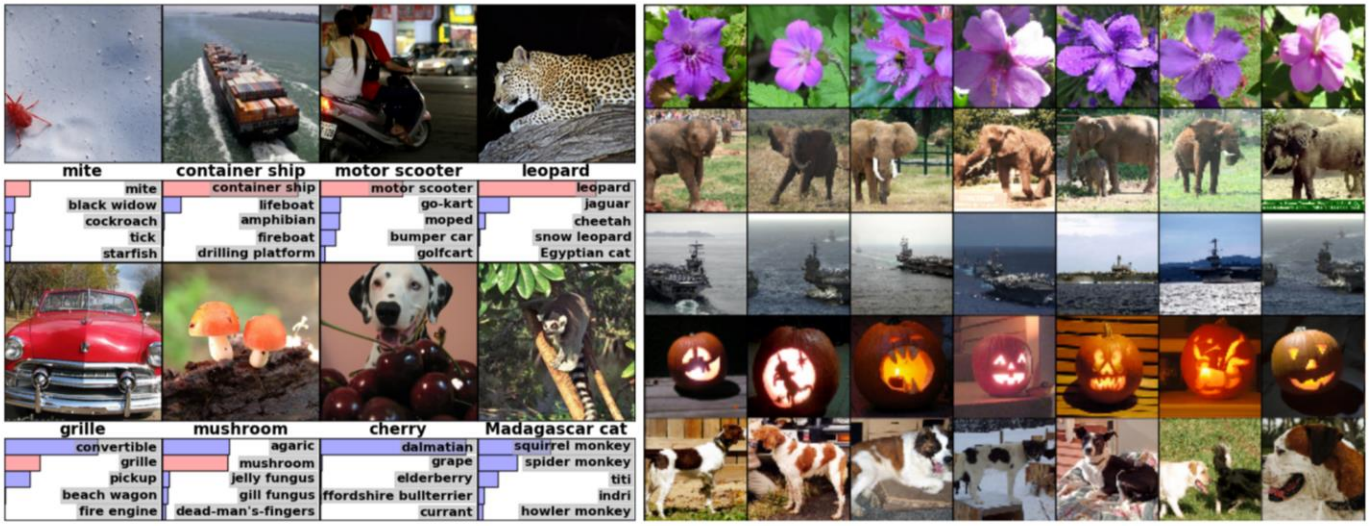


Figure 4: **(Left)** Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5). **(Right)** Five ILSVRC-2010 test images in the first column. The remaining columns show the six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.

- AlexNet can perform both **image classification** and **image retrieval**.

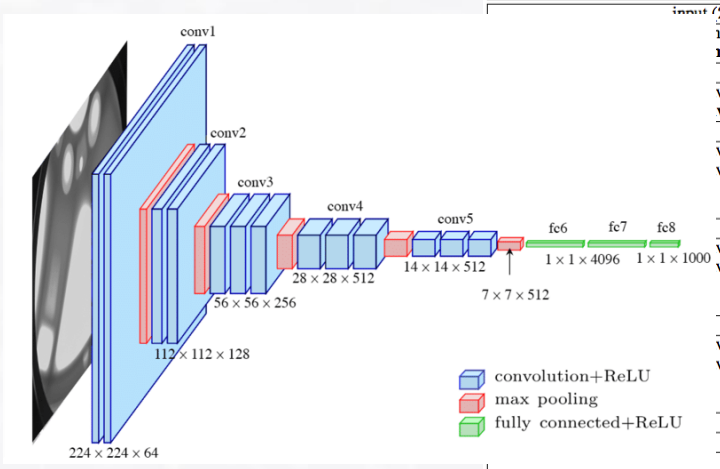
- Image retrieval is executed by extracting the penultimate feature representation (i.e., before the output layer) of dimension 4096, and retrieving the images in the dataset with feature vectors of minimum (L2) distance from the test image feature vector.

CNNs: Case Studies – VGG

VGG* (2014)

VGG-16 VGG-19

ConvNet Configuration					
A	A-LRN	B	C		
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 LRN	conv3-64	conv3-64	conv3-64	conv3-64
		conv3-64	conv3-64	conv3-64	conv3-64
maxpool					
conv3-128	conv3-128	conv3-128	conv3-128	conv3-128	conv3-128
		conv3-128	conv3-128	conv3-128	conv3-128
maxpool					
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
			conv1-256	conv3-256	conv3-256
					conv3-256
maxpool					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
			conv1-512	conv3-512	conv3-512
					conv3-512
maxpool					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
			conv1-512	conv3-512	conv3-512
					conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					



VGG16 - Structural Details

#	Input Image			output			Layer	Stride	Kernel		in	out	Param
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc		1	1	25088	4096	102764544
15	1	1	4096	1	1	4096	fc		1	1	4096	4096	16781312
16	1	1	4096	1	1	1000	fc		1	1	4096	1000	4097000
Total													138,423,208

• VGG further pushed the trend of increasing network depth (11-19 layer variants). Due to instability in training deeper models, VGG-11 was first trained, and this sub-network was used to initialize training for deeper VGG variants.

• An important innovation provided by VGG is that it **reduced filter sizes**, but increased depth (this led to performance improvements concurrent with kernel parameter savings).

• Recall that the first conv layer of AlexNet used a 7x7 filter, which requires 49 independent parameters and covers a spatial region (naturally) of size 7x7.

• By contrast, **VGG models use 3x3 filters**. Notice that if we stack three 3x3 filters in sequence we generate a nominal receptive fields equivalent to that of a 7x7 filter. However, using three 3x3 filters requires only 29 independent parameters. Conv stride for VGG is 1, maxpooling stride is 2.

*https://arxiv.org/pdf/1409.1556.pdf

**The thesis that stacking small filters in sequence captures an equivalent "effective" receptive field yielded for a large filter has recently been challenged, see: https://arxiv.org/abs/1701.04128

CNNs: Case Studies – Inception

GoogleLeNet* ("*Inception*", 2014 ILSVRC winner)

Going deeper with convolutions

Christian Szegedy
Google Inc.

Wei Liu
University of North Carolina, Chapel Hill

Yangqing Jia
Google Inc.

Pierre Sermanet
Google Inc.

Scott Reed
University of Michigan

Dragomir Anguelov
Google Inc.

Dumitru Erhan
Google Inc.

Vincent Vanhoucke
Google Inc.

Andrew Rabinovich
Google Inc.

Abstract

We propose a deep convolutional neural network architecture codenamed Inception, which was responsible for setting the new state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14). The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant. To optimize quality, the architectural decisions were based on the Hebbian principle and the intuition of multi-scale processing. One particular incarnation used in our submission for ILSVRC14 is called GoogLeNet, a 22 layers deep network, the quality of which is assessed in the context of classification and detection.

• GoogLeNet proposed a novel concept referred to as an *inception architecture* – **a network within a network**.

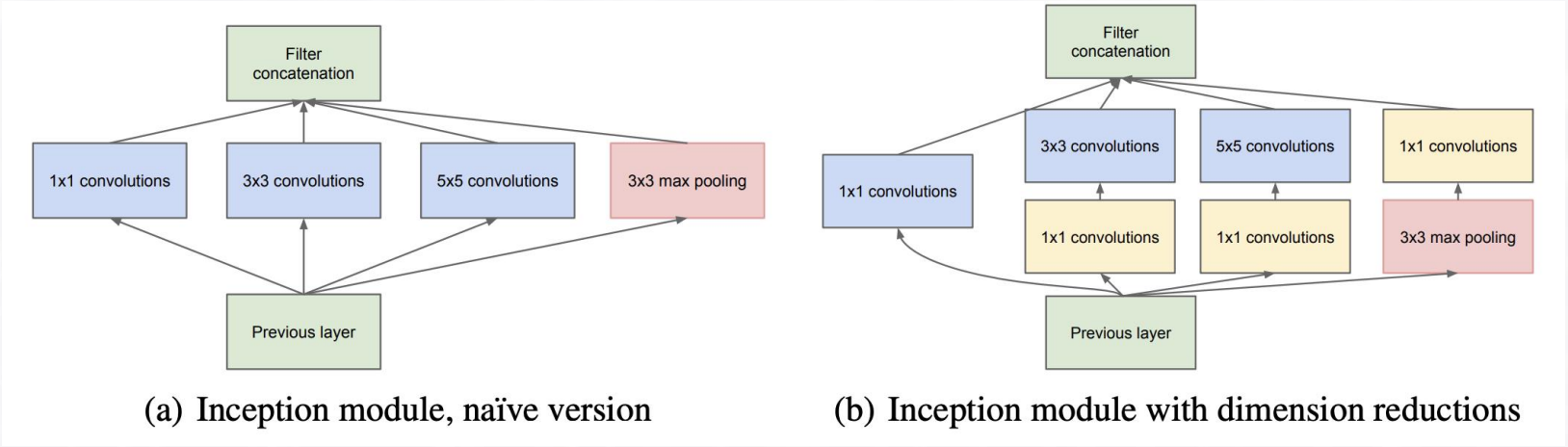
• The key device in an *inception module* is to introduce different level of spatial detail for processing within the module. A large filter will capture information in a bigger area; small filters capture details in a smaller area.

• One challenge is that we don't know *a priori* which filter sizes are best at which layer. Inception modules provide the CNN with inherent flexibility to **model different levels of image granularity in parallel**.

*https://arxiv.org/pdf/1409.4842.pdf

CNNs: Case Studies – Inception

GoogleLeNet ("*Inception*", 2014 ILSVRC winner)

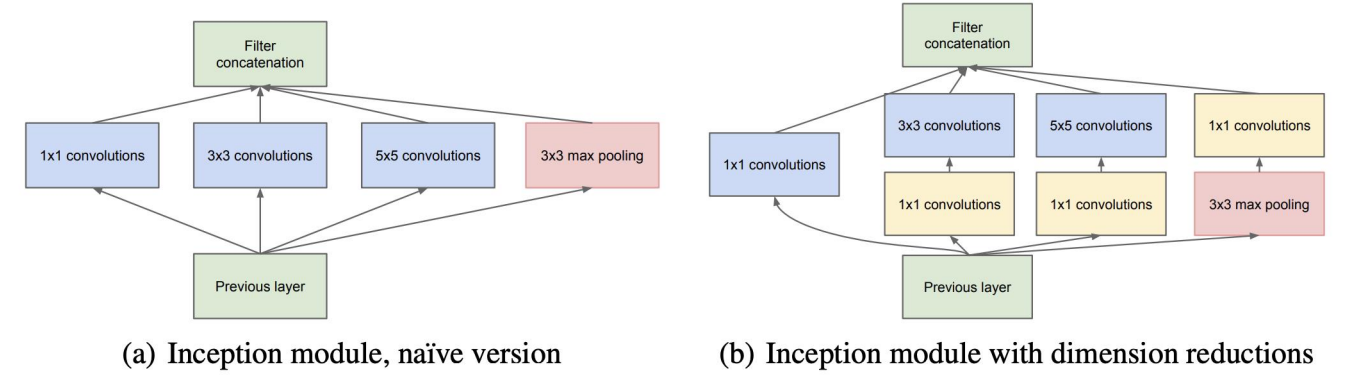


• Inception modules convolve three different filter sizes of dimension 1x1, 3x3, and 5x5 in parallel, in order to allow the network to synthesize a flexible set of spatial dimensions at once.

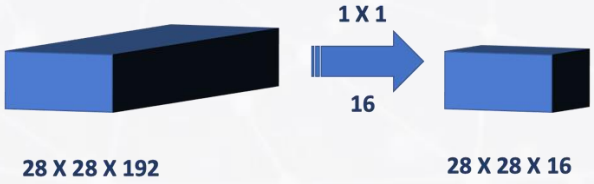
• One issue with a **naïve implementation** (see left of image) of this strategy is that it is computationally inefficient, as the number of filters at the concatenation step can be quite large (after several layers of consecutive inception modules the number of filter activations would then become unwieldy).

CNNs: Case Studies – Inception

GoogleLeNet ("*Inception*", 2014 ILSVRC winner)



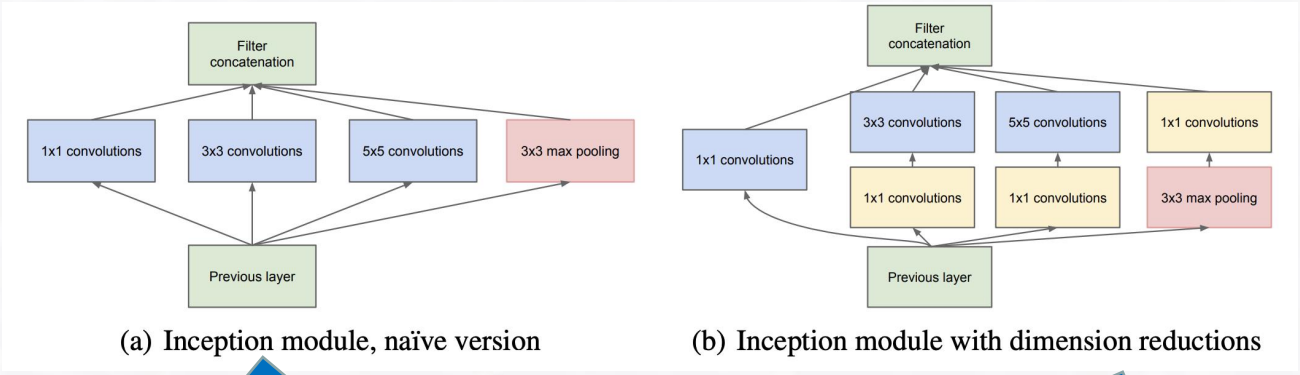
• To reduce this potential bottleneck, the authors introduce a **1x1 convolutional operation** that serves as a form of dimensionality reduction, reducing the filter depth (e.g. from 256 to 64).



• In this way, the depth of the "filter concatenation layer" (output to the inception module) can be made equal to the depth of the "previous layer" (input to the inception module), thus reducing the instance of an increase in feature maps in the network.

CNNs: Case Studies – Inception

GoogleLeNet ("*Inception*", 2014 ILSVRC winner)



(a) Inception module, naïve version

(b) Inception module with dimension reductions

```python
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers.merge import concatenate
from keras.utils import plot_model

# function for creating a naive inception block
def naive_inception_module(layer_in, f1, f2, f3):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2, (3,3), padding='same', activation='relu')(layer_in)
    # 5x5 conv
    conv5 = Conv2D(f3, (5,5), padding='same', activation='relu')(layer_in)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add inception module
layer = naive_inception_module(visible, 64, 128, 32)
```

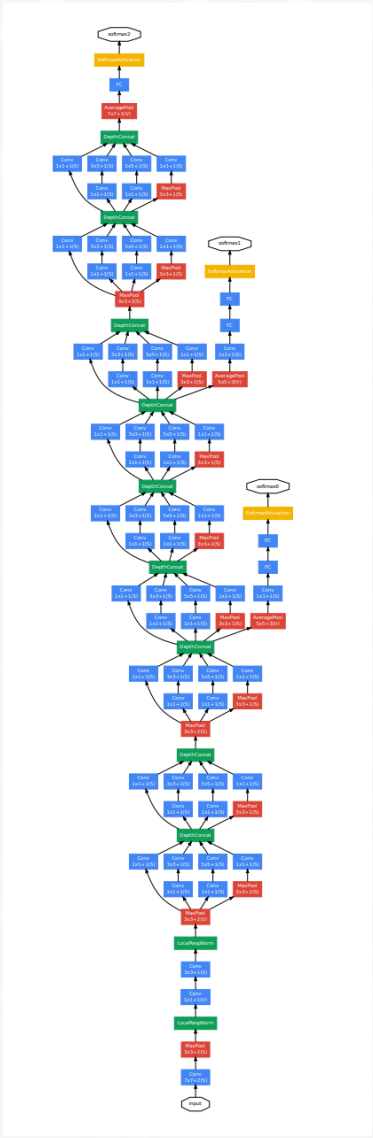
```python
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers.merge import concatenate
from keras.utils import plot_model

# function for creating a projected inception module
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
    conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
    # 5x5 conv
    conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
    conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    pool = Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add inception block 1
layer = inception_module(visible, 64, 96, 128, 16, 32, 32)
# add inception block 1
layer = inception_module(layer, 128, 128, 192, 32, 96, 64)
```

CNNs: Case Studies – Inception

GoogLeNet ("*Inception*", 2014 ILSVRC winner)



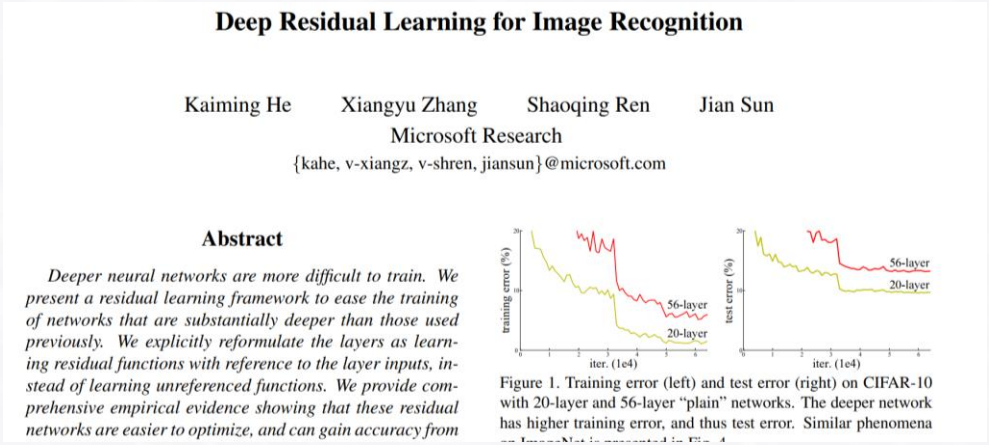
convolution
max pool
convolution
max pool
inception (3a)
inception (3b)
max pool
inception (4a)
inception (4b)
inception (4c)
inception (4d)
inception (4e)
max pool
inception (5a)
inception (5b)
avg pool
dropout (40%)
linear
softmax

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

- Inception v1 has 22 layers, ~6.7m tunable parameters.

CNNs: Case Studies – ResNet

ResNet* (Microsoft, 2015 ILSVRC winner)



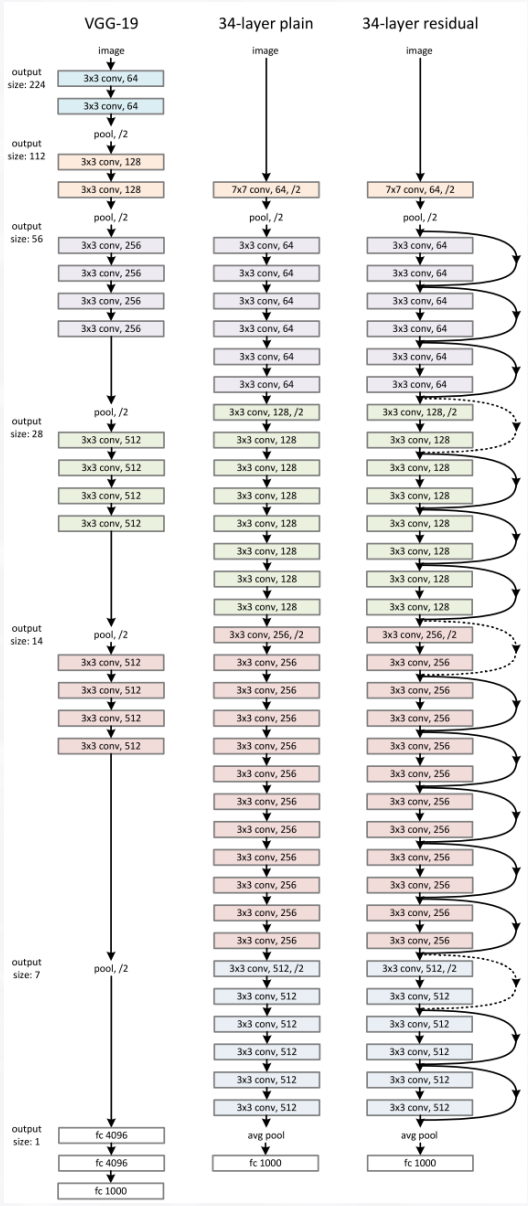
Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from

Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena



- ResNet introduces a very deep network architecture of 152 layers!

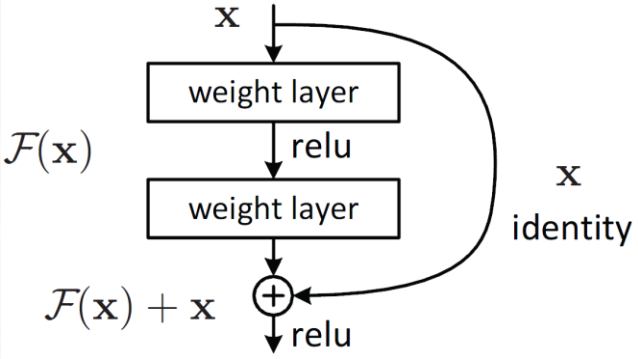
- The main challenge in training such deep networks is that gradient flow between layers is impeded by several factors, including **vanishing/exploding gradient** (this claim is somewhat controversial) and the inherent difficulty of **time-to-converge** for very large models.

- ResNet introduces **skip connections** to alleviate these difficulties.

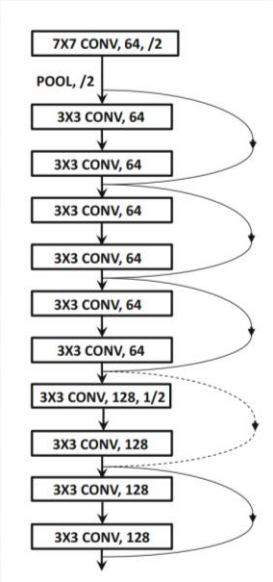
*https://arxiv.org/pdf/1512.03385.pdf

CNNs: Case Studies – ResNet

ResNet (Microsoft, 2015 ILSVRC winner)



Model	top-1 err, %	top-5 err, %	#params	time/batch 16
ResNet-50	24.01	7.02	25.6M	49
ResNet-101	22.44	6.21	44.5M	82
ResNet-152	22.16	6.16	60.2M	115

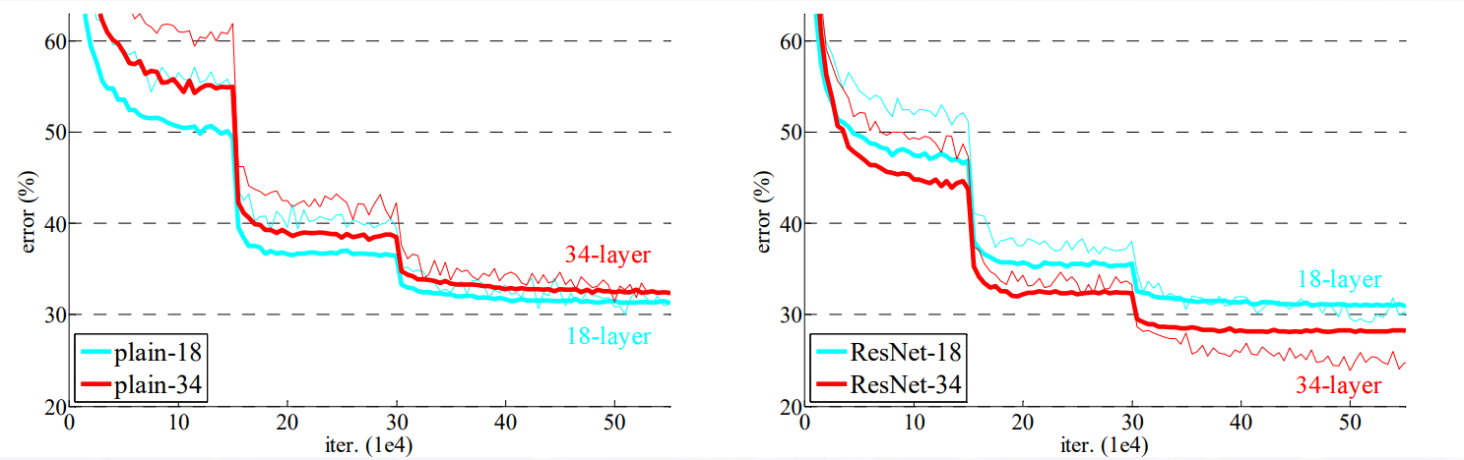


• **Skip connections** simply copy the input of layer i and adds it to layer $(i + r), r > 1$. This approach enables effective gradient flow because the backpropagation algorithm now has a "super-highway" for propagating gradients.

• This basic unit (shown on left) is called a **residual module**, and the entire network is created by putting together many of these basic modules (in some cases a 1x1 projection is applied over the skip connection to ensure dimensionality agreement).

CNNs: Case Studies – ResNet

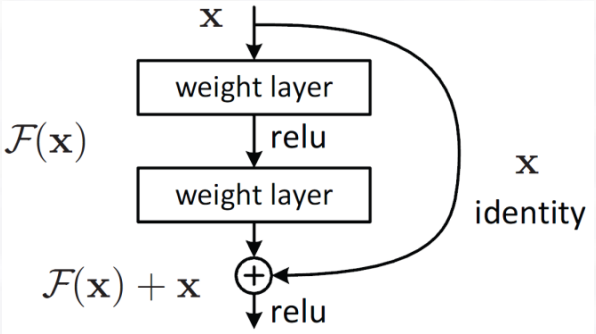
ResNet (Microsoft, 2015 ILSVRC winner)



• Experimental results for ResNet results vs. "plain" (i.e., without residual modules) CNNs; note that to help improve model training convergence, the authors reduce the learning rate by an order of magnitude every 10k epochs (approximately).

CNNs: Case Studies – ResNet

ResNet (Microsoft, 2015 ILSVRC winner)



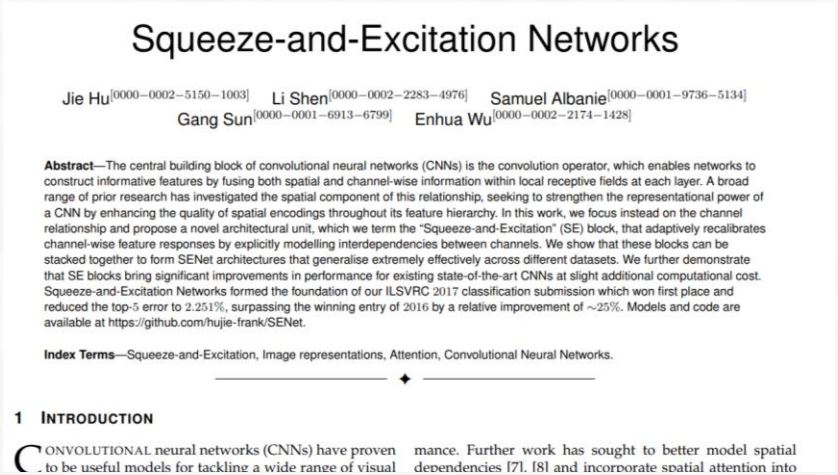

```
from keras.models import Model
from keras.layers import Input
from keras.layers import Activation
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import add
from keras.utils import plot_model

# function for creating an identity or projection residual module
def residual_module(layer_in, n_filters):
    merge_input = layer_in
    # check if the number of filters needs to be increase, assumes channels last format
    if layer_in.shape[-1] != n_filters:
        merge_input = Conv2D(n_filters, (1,1), padding='same', activation='relu', kernel_initializer='he_normal')(layer_in)
    # conv1
    conv1 = Conv2D(n_filters, (3,3), padding='same', activation='relu', kernel_initializer='he_normal')(layer_in)
    # conv2
    conv2 = Conv2D(n_filters, (3,3), padding='same', activation='linear', kernel_initializer='he_normal')(conv1)
    # add filters, assumes filters/channels last
    layer_out = add([conv2, merge_input])
    # activation function
    layer_out = Activation('relu')(layer_out)
    return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = residual_module(visible, 64)
```

CNNs: Case Studies – SE Nets

Squeeze-and-Excitation Networks* (2017 ILSVRC winner)



Squeeze-and-Excitation Networks

Jie Hu[0000−0002−5150−1003] Li Shen[0000−0002−2283−4976] Samuel Albanie[0000−0001−9736−5134]
Gang Sun[0000−0001−6913−6799] Enhua Wu[0000−0002−2174−1428]

Abstract—The central building block of convolutional neural networks (CNNs) is the convolution operator, which enables networks to construct informative features by fusing both spatial and channel-wise information within local receptive fields at each layer. A broad range of prior research has investigated the spatial component of this relationship, seeking to strengthen the representational power of a CNN by enhancing the quality of spatial encodings throughout its feature hierarchy. In this work, we focus instead on the channel relationship and propose a novel architectural unit, which we term the "Squeeze-and-Excitation" (SE) block, that adaptively recalibrates channel-wise feature responses by explicitly modelling interdependencies between channels. We show that these blocks can be stacked together to form SENet architectures that generalise extremely effectively across different datasets. We further demonstrate that SE blocks bring significant improvements in performance for existing state-of-the-art CNNs at slight additional computational cost. Squeeze-and-Excitation Networks formed the foundation of our ILSVRC 2017 classification submission which won first place and reduced the top-5 error to 2.251%, surpassing the winning entry of 2016 by a relative improvement of ∼25%. Models and code are available at https://github.com/hujie-frank/SENet.

Index Terms—Squeeze-and-Excitation, Image representations, Attention, Convolutional Neural Networks.

1 INTRODUCTION

CONVOLUTIONAL neural networks (CNNs) have proven to be useful models for tackling a wide range of visual mance. Further work has sought to better model spatial dependencies [7], [8] and incorporate spatial attention into

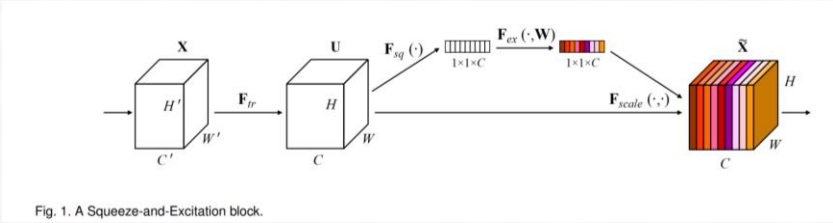


Fig. 1. A Squeeze-and-Excitation block.

• SE Nets provide a simple and elegant means to reweighting the channel outputs of convolutional layers, in order to make the processing of features more efficient in a CNN. Previous CNN architectures weight all output channels equally by default.

• The authors introduce two mechanisms to achieve this goal:

(1) **Squeeze step**: *global average pooling* is applied a feature map tensor to generate a vector representing a **global channel descriptor**.

(2) **Excitation step**: a small sub-network **recalibrates the global channel descriptor** using dynamics conditioned on the input – this allows for information-rich features to be accentuated after each conv layer in the network.

*https://arxiv.org/abs/1709.01507

CNNs: Case Studies – SE Nets

Squeeze-and-Excitation Networks (2017 ILSVRC winner)

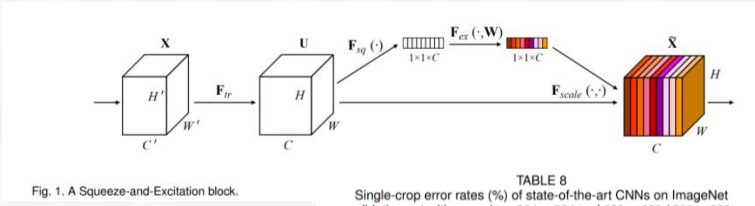
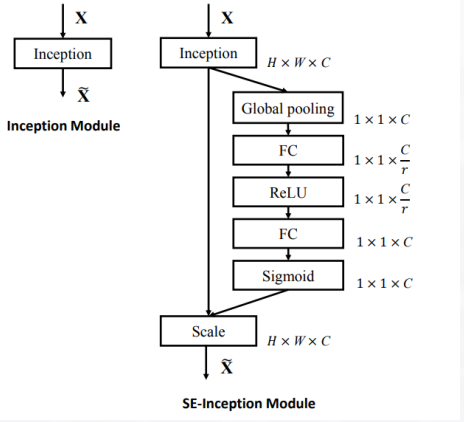
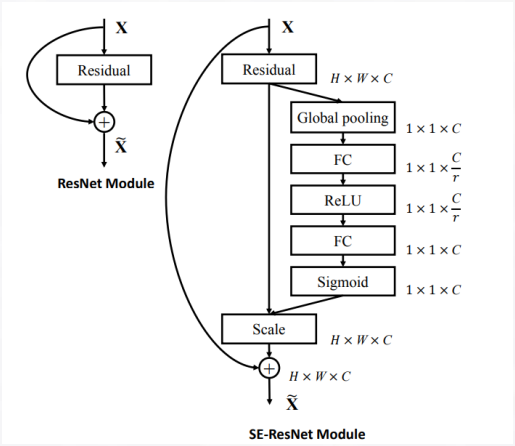


Fig. 1. A Squeeze-and-Excitation block.

TABLE 8
Single-crop error rates (%) of state-of-the-art CNNs on ImageNet validation set with crop sizes 224 × 224 and 320 × 320 / 299 × 299.

	224 × 224		320 × 320 / 299 × 299	
	top-1 err.	top-5 err.	top-1 err.	top-5 err.
ResNet-152 [13]	23.0	6.7	21.3	5.5
ResNet-200 [14]	21.7	5.8	20.1	4.8
Inception-v3 [20]	-	-	21.2	5.6
Inception-v4 [21]	-	-	20.0	5.0
Inception-ResNet-v2 [21]	-	-	19.9	4.9
ResNeXt-101 (64 × 4d) [19]	20.4	5.3	19.1	4.4
DenseNet-264 [17]	22.15	6.12	-	-
Attention-92 [58]	-	-	19.5	4.8
PyramidNet-200 [77]	20.1	5.4	19.2	4.7
DPN-131 [16]	19.93	5.12	18.55	4.16
SENet-154	**18.68**	**4.47**	**17.28**	**3.79**





- In more detail:

(1) Squeeze step: *global average pooling* is applied to the feature map tensor **U**; channel c of the tensor **U** (denoted u_c) is simply calculated as the average intensity of channel c, yielding the scalar z_c; the vector $\boldsymbol{z}$ denotes the global channel descriptor.

$$z_c = \mathbf{F}_{sq}(\mathbf{u}_c) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i,j).$$

CNNs: Case Studies – SE Nets

Squeeze-and-Excitation Networks (2017 ILSVRC winner)

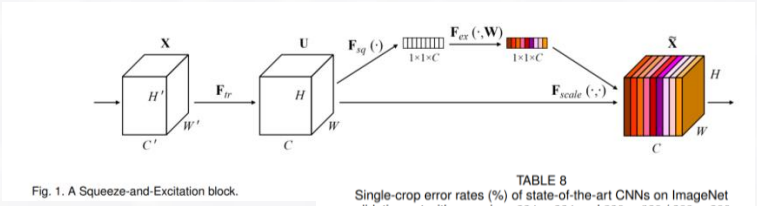
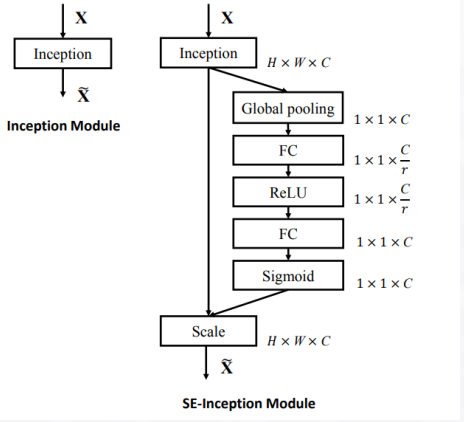
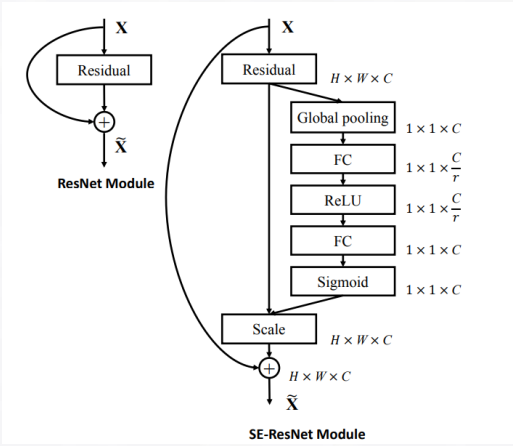


Fig. 1. A Squeeze-and-Excitation block.

TABLE 8
Single-crop error rates (%) of state-of-the-art CNNs on ImageNet validation set with crop sizes 224 × 224 and 320 × 320 / 299 × 299.

	224 × 224		320 × 320 / 299 × 299	
	top-1 err.	top-5 err.	top-1 err.	top-5 err.
ResNet-152 [13]	23.0	6.7	21.3	5.5
ResNet-200 [14]	21.7	5.8	20.1	4.8
Inception-v3 [20]	-	-	21.2	5.6
Inception-v4 [21]	-	-	20.0	5.0
Inception-ResNet-v2 [21]	-	-	19.9	4.9
ResNeXt-101 (64 × 4d) [19]	20.4	5.3	19.1	4.4
DenseNet-264 [17]	22.15	6.12	-	-
Attention-92 [58]	-	-	19.5	4.8
PyramidNet-200 [77]	20.1	5.4	19.2	4.7
DPN-131 [16]	19.93	5.12	18.55	4.16
SENet-154	**18.68**	**4.47**	**17.28**	**3.79**





- In more detail:

(1) Squeeze step: *global average pooling* is applied to the feature map tensor **U**; channel c of the tensor **U** (denoted u_c) is simply calculated as the average intensity of channel c, yielding the scalar z_c; the vector $\boldsymbol{z}$ denotes the global channel descriptor.

$$z_c = \mathbf{F}_{sq}(\mathbf{u}_c) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i,j).$$

(2) Excitation step: a small sub-network **recalibrates the global channel descriptor**. Concretely, the authors apply a sequence of a linear and non-linear operations (with the first step instantiated by $\boldsymbol{W}_1$), followed by RELU (δ), followed by an additional linear operation ($\boldsymbol{W}_2$), followed finally by a sigmoid (σ) – this particular sequence of steps additionally reduces parameter overhead through the use of 1x1 bottleneck operations.

$$\mathbf{s} = \mathbf{F}_{ex}(\mathbf{z}, \mathbf{W}) = \sigma(g(\mathbf{z}, \mathbf{W})) = \sigma(\mathbf{W}_2 \delta(\mathbf{W}_1 \mathbf{z})),$$

CNNs: Effects of Depth

• As we have seen, many of the significant advances in performance in recent years in the ILSVRC contest are a result of improved computational power, greater data availability, and changes in architectural design that have enabled the effective training of NNs with **increased depth**.

Name	Year	Number of Layers	Top-5 Error
–	Before 2012	≤ 5	$> 25\%$
AlexNet	2012	8	15.4%
ZfNet/Clarifai	2013	$8/> 8$	14.8% / 11.1%
VGG	2014	19	7.3%
GoogLeNet	2014	22	6.7%
ResNet	2015	152	3.6%

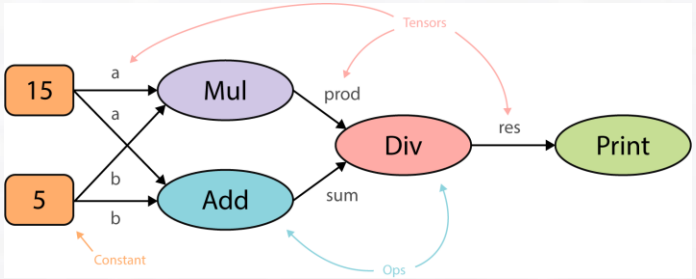
• The rapid increase in accuracy in the short period from 2012-2015 is quite remarkable and is unusual for most ML applications that are as well studies as image classification.

• Increased depth of the NN is closely correlated with improved error rates. As such, as important focus of research in recent years have been to enable algorithmic modifications that support increased depth of NNs.

Open-Source DL Frameworks

• **TensorFlow** (Google, 2015): Python, C++, Java, etc., library for DL; ML algos represented as computation graph.

https://www.tensorflow.org/



• **Pytorch** (FAIR, 2016): Python framework based on the *Torch library* (similar to NumPy, but used for tensors with GPU acceleration).

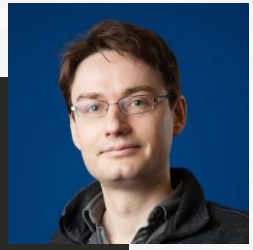
https://pytorch.org/

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Assuming that we are on a CUDA machine, this should print a CUDA device:
```

• **Keras** (Chollet, Google): open source (Python) interface for TF, GPU acceleration support.

https://keras.io/



```
from tensorflow import keras
from tensorflow.keras import layers

# Instantiate a trained vision model
vision_model = keras.applications.ResNet50()

# This is our video encoding branch using the trained vision_model
video_input = keras.Input(shape=(100, None, None, 3))
encoded_frame_sequence = layers.TimeDistributed(vision_model)(video_input)
encoded_video = layers.LSTM(256)(encoded_frame_sequence)
```

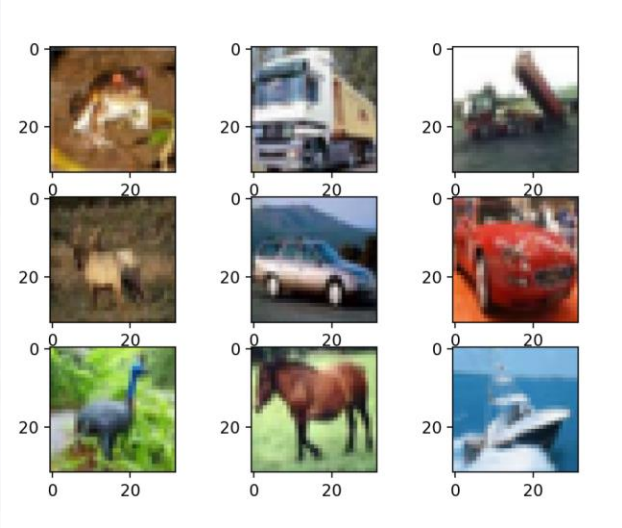
Open-Source DL Frameworks

• Next, we develop a hands-on (Keras) exampled of training a CNN-based classifier on the CIFAR-10 dataset.

(1) Load and prepare dataset

```
# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = cifar10.load_data()
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY
```

```
# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm
```

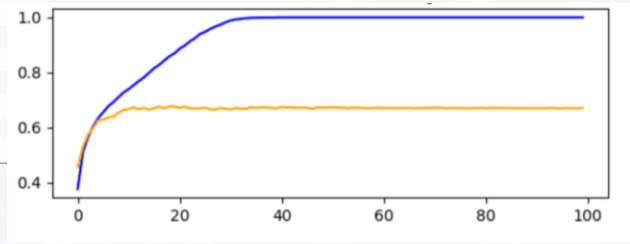


Open-Source DL Frameworks

• Next, we develop a hands-on (Keras) exampled of training a CNN-based classifier on the CIFAR-10 dataset.

(2a) Train a "VGG-type" CNN, using one VGG block

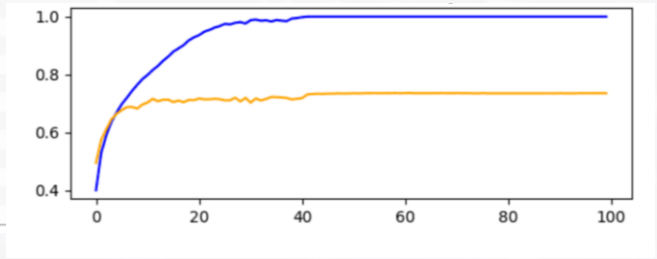
```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```



VGG1: 67.1% Test Accuracy

(2b) Train a "VGG-type" CNN, using three VGG blocks

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

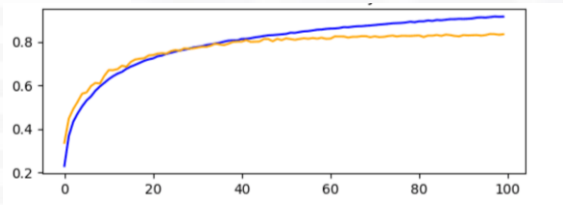


VGG2: 73.5% Test Accuracy

Open-Source DL Frameworks

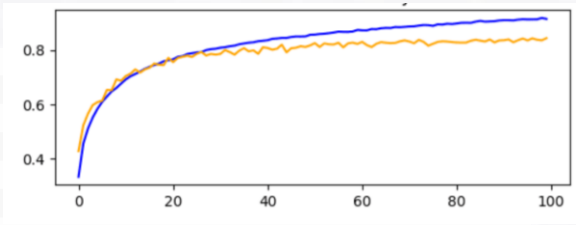
(3a) Train a "VGG-type" CNN with Dropout regularization

```
define cnn model
ef define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dropout(0.2))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```



VGG w/Dropout: 83.5% Test Accuracy

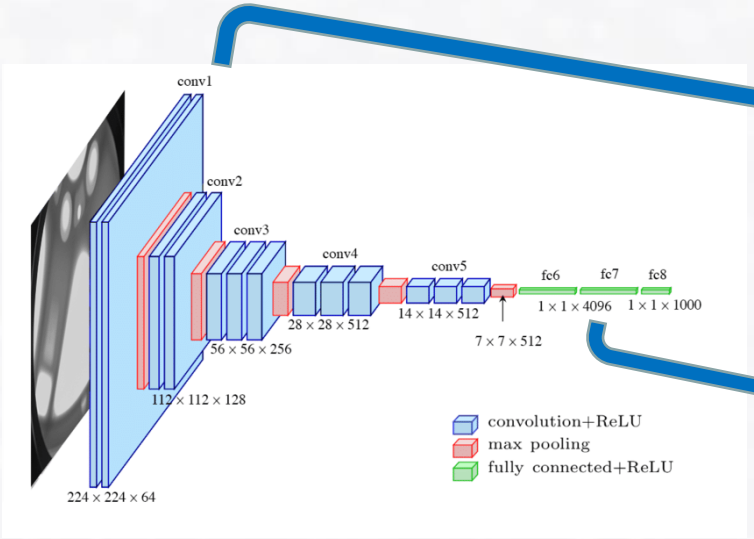
(3b) Train a "VGG-type" CNN, using data augmentation

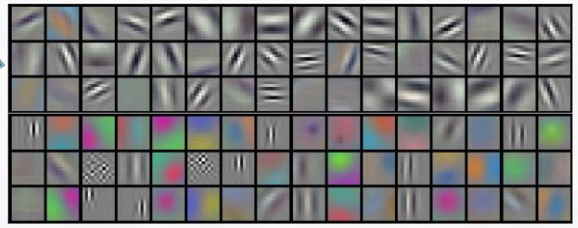


VGG w/DA: 84.5% Test Accuracy

```
# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
    # prepare iterator
    it_train = datagen.flow(trainX, trainY, batch_size=64)
    # fit model
    steps = int(trainX.shape[0] / 64)
    history = model.fit_generator(it_train, steps_per_epoch=steps, epochs=100, validation_data=(testX, testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
```

CNNs: Pre-trained Models





Pre-trained CNN features (extracted from conv1)

$\in \mathbb{R}^{4096}$

Latent image representation (extracted from FC7)

• A key point about image data is that the extracted features from a particular data set are highly reusable across data sources.

• Thus, features extracted from, for example, a CNN trained on ImageNet, can often be reused directly in a new model (cf. transfer learning), or, conversely, a compact latent representation of the input image data (typically from the penultimate network layer), can be extracted as a meaningful encoding of an image.

• In some cases, even all (or most) of the features from a pre-trained CNN are repurposed in a new model.

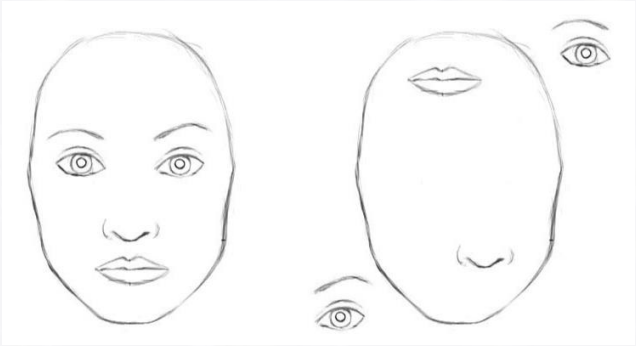
• Using a pretrained model naturally can save time/compute, and often obviates the need to train a CNN from scratch.

Limitations of CNNs

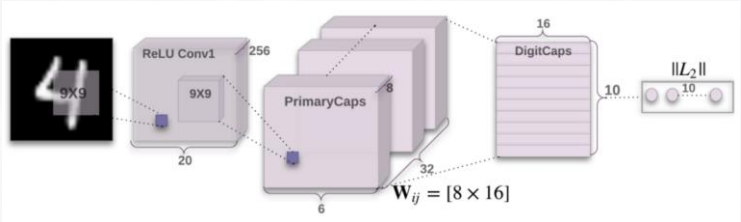
•According to Geoffrey Hinton*, it is unfortunate in some sense that CNNs work so well, because they have serious flaws which he believes "will be hard to get rid of."

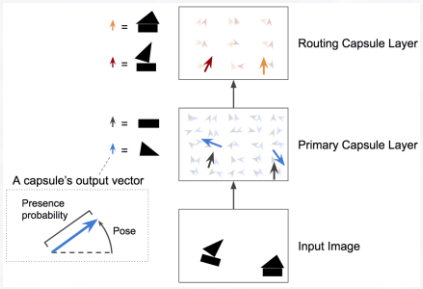
These flaws include (according to Hinton):

- **Inefficiencies in backpropagation** paradigm itself
- **Poor translational invariance**
- **Lack of "pose" information** – absence of nuanced information about relative position and orientation of parts of an object. These is sometimes referred to as the "Picasso problem."



- Hinton proposed **Capsule Nets**** (2017) to address some of these issues.



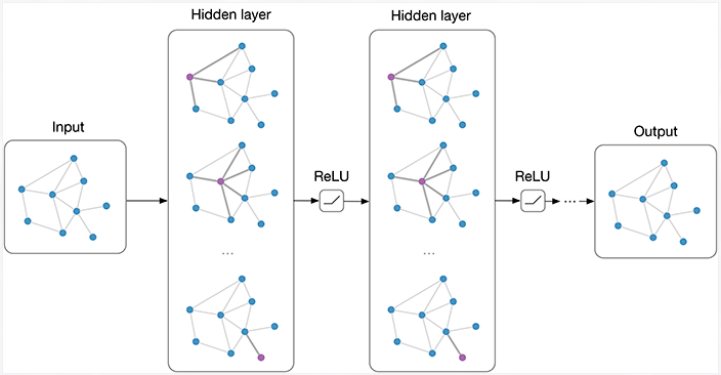


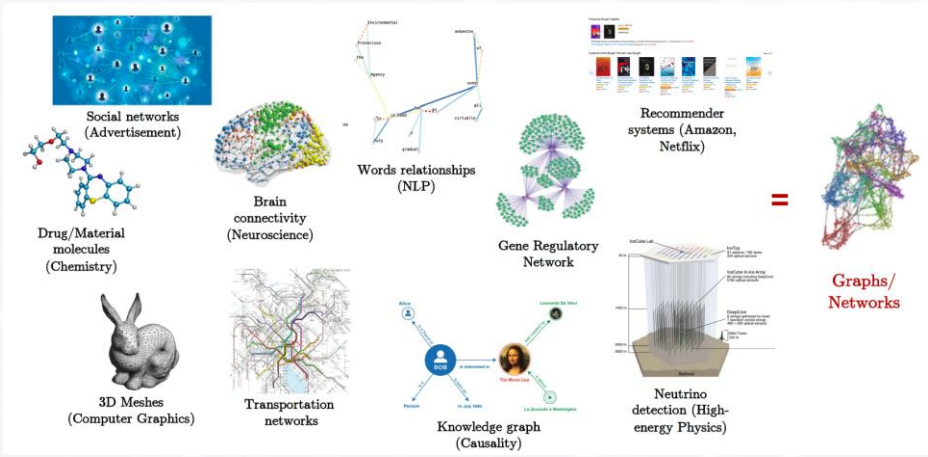
*https://www.youtube.com/watch?v=rTawFwUvnLE&t=8s

** https://arxiv.org/abs/1710.09829

Limitations of CNNs

•Beyond these issues, traditional CNNs are also ill-equipped to efficiently process graphical data, particularly data with non-uniform neighborhood information.





• **Graph Neural Networks*** (GNNs) and **Graph Convolutional Neural Networks** (GCNs) are specifically aimed at overcoming these deficiencies *vis-à-vis* traditional CNNs.

*https://arxiv.org/pdf/1812.08434.pdf

Fin