

Machine Learning

Convolutional Neural Networks

Gabriele Russo Russo Francesco Lo Presti

Laurea Magistrale in Ingegneria Informatica – A.Y. 2023/24



TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria Civile e
Ingegneria Informatica

Motivation: Image Classification

- ▶ Suppose we are asked to classify images of cats and dogs
- ▶ Consider a dataset of 1-megapixel labeled photographs
 - ▶ Each input instance has 1 million pixels
- ▶ Even an aggressive reduction to 1,000 hidden units would require a fully connected layer with $10^6 \times 10^3 = 10^9$ parameters
 - ▶ Training would be extremely difficult or even unfeasible

Motivation (2)

- ▶ Recall the “Fashion MINST” example, where we classified images using a NN
- ▶ Image = 2D grid of pixels
- ▶ We ignored this structure and *flattened* images into vectors
 - ▶ We could even shuffle the vector elements, as the NN was invariant to feature order
- ▶ How to leverage prior knowledge that nearby pixels are typically related to each other?

Convolutional Neural Networks (CNN)

LeCun et al., "Backpropagation applied to handwritten zip code recognition". Neural Computation (1989)

- ▶ **Convolutional Neural Networks (CNN)**: specialized NNs for processing data that has a known grid-like topology, e.g.:
 - ▶ time-series data (1D topology)
 - ▶ images (2D grids of pixels)
- ▶ "Convolutional" because of the **convolution** operation used in the NN
- ▶ A CNN is a neural network that uses convolution in at least one layer

Key Ideas

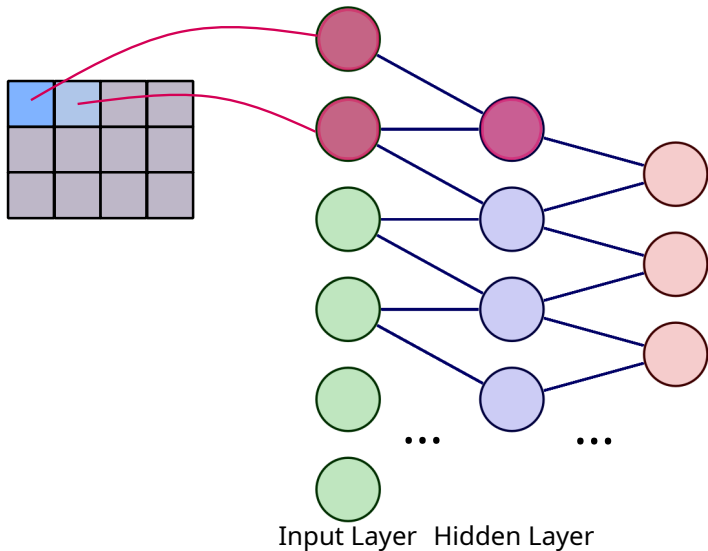
CNNs leverages 3 important ideas to overcome the limitations mentioned above:

- ▶ sparse interactions
- ▶ parameter sharing
- ▶ equivariant representations

Sparse Interactions

- ▶ So far, we have considered *fully connected* layers
 - ▶ every unit interacts with every input unit
- ▶ With a large number of units (as in the motivating example at the beginning), this approach leads to a huge number of parameters!
- ▶ **Idea:** let each unit interact with a reduced portion of the input

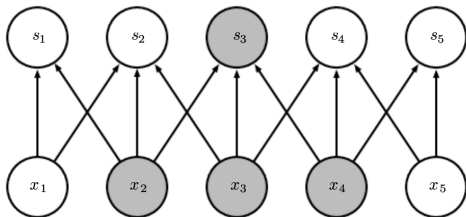
Example: Sparse Connections



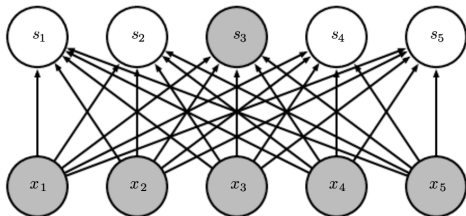
Connections & Receptive Field

- ▶ With sparse connections, it seems that we lose the ability of detecting relevant interactions between distant portions of the input
- ▶ Let's consider the **receptive field** of each unit: the set of inputs that impact the output of a given unit

Example: Receptive Field

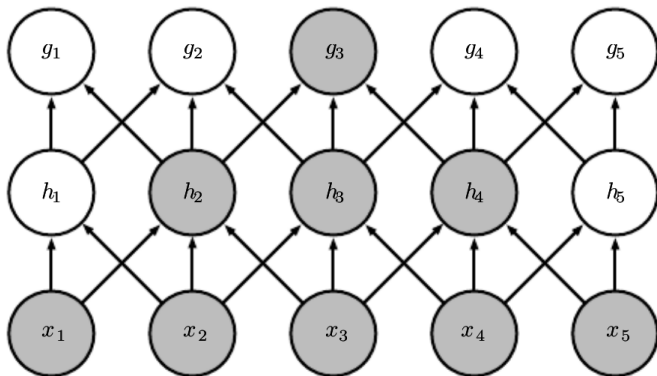


Sparse
Connections



Fully Connected

Example: Receptive Field with More Layers

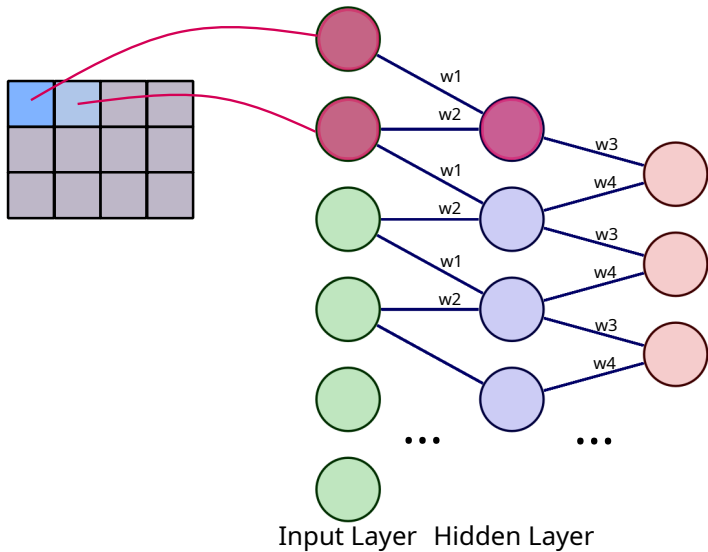


Despite sparsity of *direct* connections, units in higher layers *indirectly* interact with all or most of the input

Parameter Sharing

- ▶ In a traditional NN, each pair of connected units has its own weight w_{ij}
- ▶ Convolutional layers re-use the same set of parameters to compute the output of each unit
 - ▶ the same parameters are **shared** across all the units
- ▶ While the complexity of backpropagation does not change, we have important benefits
- ▶ Reduced memory requirement!
- ▶ Reduced number of parameters to learn!

Example

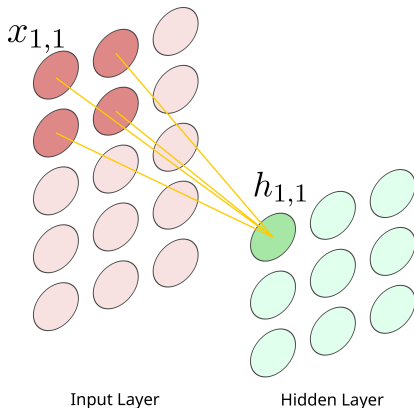


Equivariant Representation

- ▶ Parameter sharing (as defined above) causes the layer to have a property called **equivariance to translation**
- ▶ Because the same parameters are used across different parts of the input, if we move an object in the input, its representation will move the same amount in the output
- ▶ A traditional NN can completely fail if we change the order in which features appear!
- ▶ Clearly, parameter sharing does not make CNNs equivariant to some other transformations, such as changes in the scale or rotation of an image

Towards Convolutional Layers

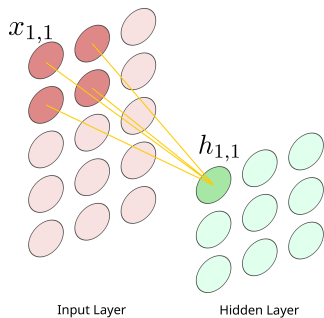
- ▶ When working with grid data (e.g., images), it is useful to think of hidden units being arranged in a grid as well
- ▶ Each unit still connected to a small portion of the input



Towards Convolutional Layers (2)

$$h_{1,1} = k_{1,1}x_{1,1} + k_{1,2}x_{1,2} + k_{2,1}x_{2,1} + k_{2,2}x_{2,2}$$

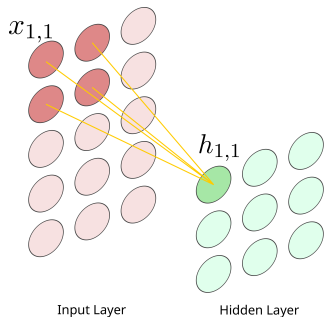
$$h_{i,j} = k_{1,1}x_{i,j} + k_{1,2}x_{i,j+1} + k_{2,1}x_{i+1,j} + k_{2,2}x_{j+1,i+1}$$



- ▶ Params $k_{i,j}$ form matrix K (similar to the W used in fully connected layers)
- ▶ Called **kernel** or **filter**

Example

$$K = \begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}$$



- ▶ What are we computing?
- ▶ We are just averaging the value of 4 neighboring pixels

Convolution

- ▶ The operation of convolutional layers can be mathematically described as a convolution applied to the **input** tensor and the **kernel**
- ▶ E.g., for a 2D image X as input, we use a 2D kernel K

$$(X * K)(i, j) = \sum_a \sum_b K(a, b) X(i + a, j + b)$$

Convolution vs. Cross-Correlation

- ▶ Actually, convolution on discrete data is defined as:

$$S(i, j) = \sum_a \sum_b K(a, b) X(i - a, j - b)$$

- ▶ Most the ML libraries use the term “convolution” but actually implement **cross-correlation**

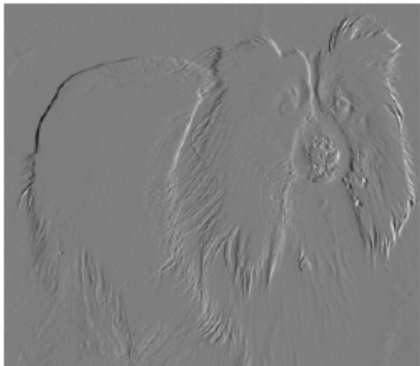
$$S(i, j) = \sum_a \sum_b K(a, b) X(i + a, j + b)$$

- ▶ Besides losing commutative property with cross-correlation, there is **no impact** in practice because we are going to *learn* the kernel (flipped or not)

Basic Convolutional Layer

- ▶ Let's use convolution to construct a hidden layer
- ▶ A convolutional layer takes a tensor \mathbf{X} in input and produces an output tensor \mathbf{H}
- ▶ \mathbf{H} computed by calculating convolution $\forall(i, j)$
 - ▶ We are skipping many details at this point, including the use of a nonlinear activation function
- ▶ The kernel \mathbf{K} comprises the parameters of the layer (to be trained)

Example: Detecting Vertical Edges



A 2-pixel kernel is enough: $K = [1, -1]$

Example

Input X

0	1	2
3	4	5
6	7	8

Kernel K

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

H

$$\begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix}$$

- ▶ We started with a 3x3 input and got a 2x2 output
- ▶ What if we want to stack N convolutional layers??

Zero Padding

- ▶ To solve this problem, we can add extra pixels of filler around the boundary of our input (**padding**)
- ▶ Typically, we set the values of the extra pixels to zero

Input Kernel Output

0	0	0	0	0
0	0	1	2	0
0	3	4	5	0
0	6	7	8	0
0	0	0	0	0

*

0	1
2	3

=

0	3	8	4
9	19	25	10
21	37	43	16
6	7	8	0

Zero Padding (2)

- ▶ With input of size $n_h \times n_w$ and kernel of size $k_h \times k_w$, output has shape $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- ▶ Adding p_h rows and p_w columns of padding, we get:
 $(n_h + p_h - k_h + 1) \times (n_w + p_w - k_w + 1)$
- ▶ To give input and output the same size, usually we set $p_h = k_h - 1$ and $p_w = k_w - 1$
- ▶ CNN kernels usually have **odd** width and height
 - ▶ If k_h is odd, we exactly add $\frac{p_h}{2}$ rows both on top and bottom (similarly for columns)
 - ▶ As an additional benefit, the output element \mathbf{H}_{ij} is calculated with the convolution window centered on \mathbf{X}_{ij}

Example (with padding)

Input (3x4)

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 0 & 0 \\ 7 & 8 & 9 & 1 \end{bmatrix}$$

With Padding

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 2 & 3 & 4 & 0 \\ 0 & 2 & 5 & 0 & 0 & 0 \\ 0 & 7 & 8 & 9 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Kernel K

$$\begin{bmatrix} a & b & c \\ c & a & b \\ e & c & f \end{bmatrix}$$

$$(K * I)(1, 1) = \dots$$

Convolution as Matrix Multiplication

- ▶ Convolution (and cross-correlation) can be regarded as traditional matrix multiplication
- ▶ The matrices to multiply must be constructed from the original input and kernel
- ▶ For instance, consider the following input A and kernel K (without padding):

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad K = \begin{bmatrix} k_{1,1} & k_{1,2} \\ k_{2,1} & k_{2,2} \end{bmatrix}$$

We construct a matrix M from K and a vector v from A

$$M = \begin{bmatrix} k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} \end{bmatrix}$$

$$v = [a_{1,1} \quad a_{1,2} \quad a_{1,3} \quad a_{2,1} \quad a_{2,2} \quad a_{2,3} \quad a_{3,1} \quad a_{3,2} \quad a_{3,3}]$$

$$Mv^T = \begin{bmatrix} a_{1,1}k_{1,1} + a_{1,2}k_{1,2} + a_{1,3} \cdot 0 + a_{2,1}k_{2,1} + a_{2,2}k_{2,2} + a_{2,3} \cdot 0 + a_{3,1} \cdot 0 + a_{3,2} \cdot 0 + a_{3,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2}k_{1,1} + a_{1,3}k_{1,2} + a_{2,1} \cdot 0 + a_{2,2}k_{2,1} + a_{2,3}k_{2,2} + a_{3,1} \cdot 0 + a_{3,2} \cdot 0 + a_{3,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2} \cdot 0 + a_{1,3} \cdot 0 + a_{2,1}k_{1,1} + a_{2,2}k_{1,2} + a_{2,3} \cdot 0 + a_{3,1}k_{2,1} + a_{3,2}k_{2,2} + a_{2,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2} \cdot 0 + a_{1,3} \cdot 0 + a_{2,1} \cdot 0 + a_{2,2}k_{1,1} + a_{2,3}k_{1,2} + a_{3,1} \cdot 0 + a_{3,2}k_{2,1} + a_{3,3}k_{2,2} \end{bmatrix}$$

Reshaping into a 2x2 matrix we get the result:

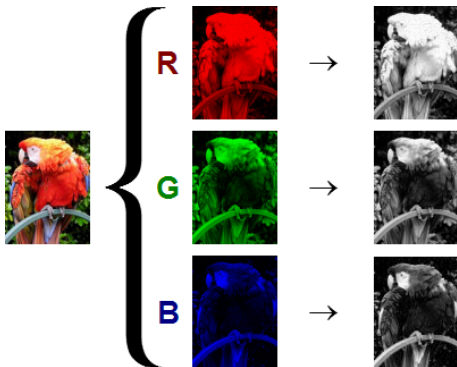
$$\begin{bmatrix} a_{1,1}k_{1,1} + a_{1,2}k_{1,2} + a_{2,1}k_{2,1} + a_{2,2}k_{2,2} & a_{1,2}k_{1,1} + a_{1,3}k_{1,2} + a_{2,2}k_{2,1} + a_{2,3}k_{2,2} \\ a_{2,1}k_{1,1} + a_{2,2}k_{1,2} + a_{3,1}k_{2,1} + a_{3,2}k_{2,2} & a_{2,2}k_{1,1} + a_{2,3}k_{1,2} + a_{3,2}k_{2,1} + a_{3,3}k_{2,2} \end{bmatrix}$$

Convolution as Matrix Multiplication (2)

- ▶ Performing convolution as matrix multiplication does not look much more efficient (larger matrices are involved!)
- ▶ However, this interpretation shows that conceptually convolutional layers are not different than fully connected layers, in terms of mathematical operations involved
- ▶ Therefore, we are guaranteed that backpropagation can still be used to compute gradients!

Channels

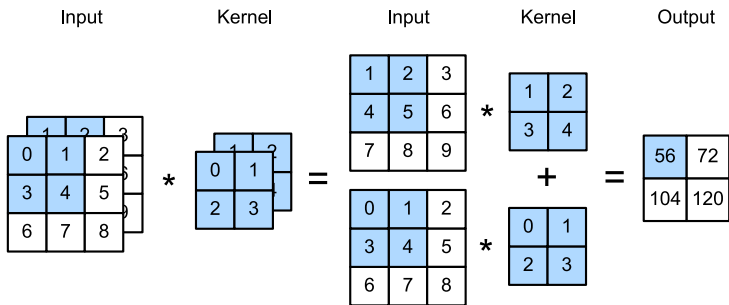
- ▶ Images usually consist of multiple **channels**, usually three: red, green and blue (RGB)
- ▶ Represented as 3D tensors with shape: $3 \times h \times w$



Multiple Input Channels

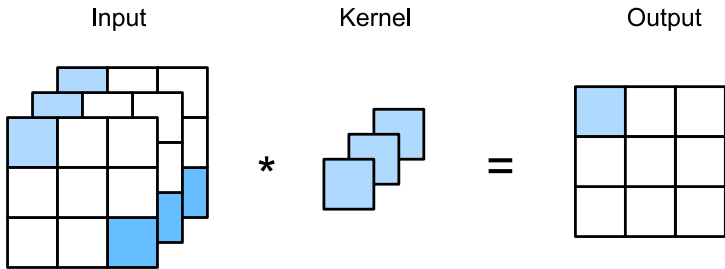
With $c_i > 1$ input channels, we need a 3D kernel as well (but we still get a 2D output tensor)

$$H_{i,j} = \sum_a \sum_b \sum_c K_{a,b,c} X_{i+a,j+b,c}$$



Example: 1x1 kernel

- ▶ Apparently, a 1×1 kernel does not make much sense, as it cannot correlate any pixels!
- ▶ With multi-channel input, a $1 \times 1 \times c_i$ kernel can be actually useful
 - ▶ e.g., to average the value of a pixel across the channels



Multiple Output Channels

- ▶ Regardless of the number of input channels, so far we always ended up with 1 output channel
- ▶ In practice, a convolutional layer applies multiple kernels at the same time, each leading to an output channel
 - ▶ Different kernels can discover different things in the input
- ▶ Having c_o kernels with shape $k_h \times k_w \times c_i$, equivalent to a single kernel with shape $k_h \times k_w \times c_i \times c_o$

$$H_{i,j,k} = \sum_a \sum_b \sum_c K_{a,b,c,k} X_{i+a,j+b,c}$$

Convolutional Layer: Revisited

- ▶ Tensor \mathbf{X} in input, with shape $n_h \times n_w \times c_i$
- ▶ Kernel \mathbf{K} with shape $k_h \times k_w \times c_i \times c_o$ comprises the parameters of the layer
 - ▶ Plus a bias term for every kernel
- ▶ Output tensor \mathbf{H} with shape $n_h \times n_w \times c_o$
 - ▶ Assuming zero padding is used as described above
- ▶ \mathbf{H} computed by calculating convolution with \mathbf{K} at every input position, and possibly applying a nonlinear activation function (e.g., ReLU)

$$H_{i,j,k} = \phi \left(\sum_{u,v,c} K_{u,v,c,k} X_{i+u,j+v,c} + b_k \right)$$

where b_k is the bias term for the k -th output channel.

Convolution with Stride

- ▶ When computing cross-correlation, the convolution window starts from the upper-left corner of the input, and then slides over all locations down and to the right
 - ▶ So far, we defaulted to sliding one element at a time
- ▶ Sometimes we prefer to move the window more than one element at a time, skipping intermediate locations
 - ▶ Computational efficiency
 - ▶ Downsampling the input image
- ▶ **Stride** = number of rows and columns traversed per slide
 - ▶ e.g., with stride 2, we halve both width and height of the image

Example

Kernel K

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

Stride 1

$$\begin{bmatrix} 0 & 1 & 2 & 7 \\ 4 & 3 & 5 & 0 \\ 6 & 7 & 8 & 2 \end{bmatrix}$$

Stride 2

$$\begin{bmatrix} 0 & 1 & 2 & 7 \\ 4 & 3 & 5 & 0 \\ 6 & 7 & 8 & 2 \end{bmatrix}$$

Pooling

- ▶ Convolutional layers are usually followed by **pooling layers**
 - ▶ Some books just consider convolution and pooling distinct steps of the same “complex” layer
- ▶ Pooling operator: fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed
- ▶ Similar to convolution, but with no parameters!
- ▶ Pooling operators are deterministic, typically calculating either the **maximum or the average** value of the elements in the pooling window
 - ▶ “max pooling” and “average pooling” (for short)

Example

Input

0	1	2
3	4	5
6	7	8

2 x 2
Max-pooling

Output

4	5
7	8

Pooling (2)

- ▶ In presence of multiple channels, pooling is applied to each channel separately
 - ▶ Output tensor has the same number of channels as the input
- ▶ Since pooling aggregates information from an area, it is frequent to match pooling window size and stride
 - ▶ with a 3×3 window, stride is set to 3

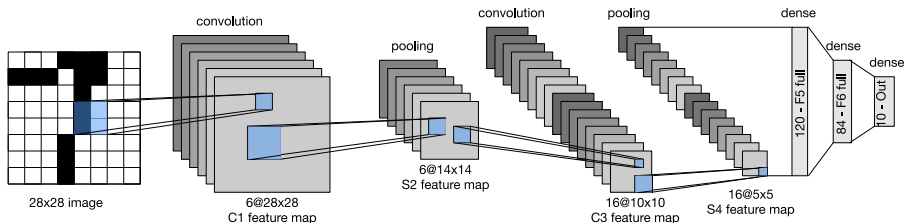
LeNet-5 (1998)


- ▶ We are now ready to assemble a fully-functional CNN
- ▶ **LeNet-5** (or simply, LeNet) is one of the first published CNNs
 - ▶ Introduced by (and named for) Yann LeCun, after a decade of research on CNNs
 - ▶ Received wide attention for its performance
- ▶ Initially aimed at recognizing handwritten digits
 - ▶ Matched the performance of support vector machines, then a dominant approach in supervised learning, achieving an error rate of less than 1% per digit
 - ▶ Eventually adapted to recognize digits for deposits in ATM machines (still used in some ATMs today!)

LeNet: Architecture

At a high level, LeNet consists of two parts:

- ▶ a convolutional encoder consisting of 2 convolutional layers
- ▶ a dense block consisting of 3 fully connected layers



 `tf_lenet.ipynb`

Example: LeNet and Fashion MNIST

- ▶ We consider again the Fashion MNIST dataset we have used in the previous lectures
- ▶ Instead of using a MLP, this time we use LeNet

 `tf_lenet_fashion.ipynb`

Beyond LeNet

- ▶ Most of the modern successful CNN models deeply inspired by LeNet-5
- ▶ However, after LeNet publication in 1998, no major advancements until 2010s
 - ▶ Waiting for more data and hardware!
- ▶ **ImageNet Large Scale Visual Recognition Challenge¹**:
barometer of progress on supervised learning in computer vision since 2010
 - ▶ From 27% to 2% of error in 5-6 years
 - ▶ We review some CNN architectures that were winners or runners-up

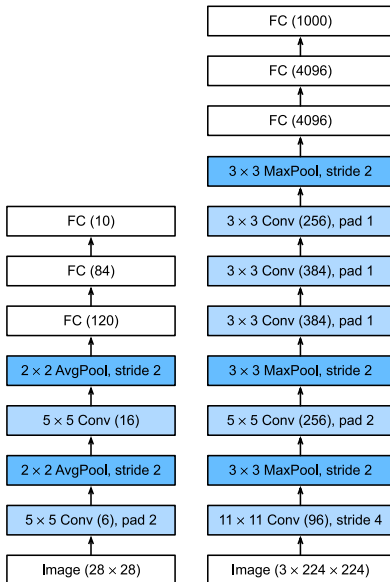
¹<https://www.image-net.org/challenges/LSVRC/index.php>

AlexNet (2012)

- ▶ First deep CNN to outperform traditional supervised learning approaches
- ▶ Won the ImageNet competition in 2012 by a large margin
- ▶ Architecture similar to LeNet, with important differences:
 - ▶ Deeper (8 layers)
 - ▶ Larger layers (e.g., dense layers require 160 MB of weights)
 - ▶ Larger and multi-channel input images
 - ▶ ReLU replaces sigmoid
 - ▶ Dropout for regularization

 `tf_alexnet_fashion.ipynb` (GPU recommended!)

AlexNet vs. LeNet

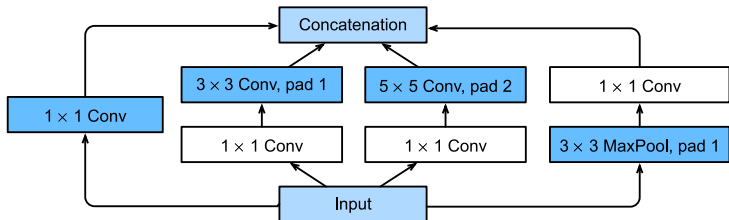


GoogLeNet (2014)

- ▶ Winner of the ImageNet competition in 2014
- ▶ Clear distinction of:
 - ▶ Stem (data ingestion): first 2-3 conv. layers
 - ▶ Body (data processing): other convolutional blocks
 - ▶ Head (prediction)
- ▶ We see a slightly simplified version of GoogLeNet: the original design included a number of tricks for stabilizing training, no longer necessary due to the availability of improved training algorithms

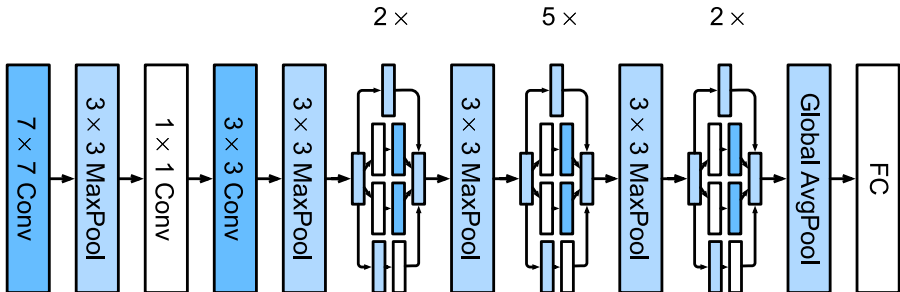
GoogLeNet: Inception Block

- ▶ Key contribution of GoogLeNet
- ▶ Other works tried to identify the best convolution kernel (from 1×1 to 11×11)
- ▶ GoogLeNet simply concatenates multi-branch convolutions (Inception block)



GoogLeNet: Architecture

- ▶ A total of 9 Inception blocks
- ▶ *Global pooling* at the end to reduce each channel to a single value
- ▶ Fewer parameters than AlexNet

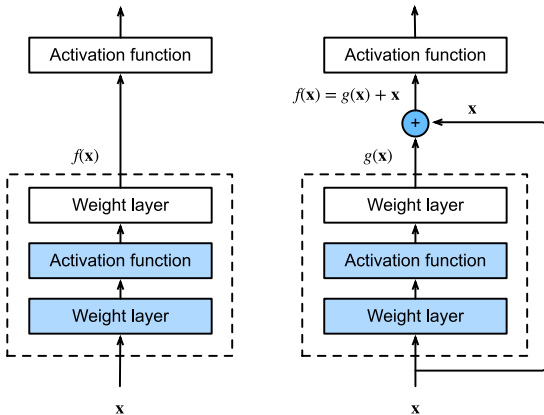


ResNet (2015)

- ▶ Winner of the ImageNet competition in 2015
- ▶ A deeper model (100+), with fewer parameters
 - ▶ Trend already started in previous years
- ▶ Key element: **skip connections** (or, **shortcut connections**)
 - ▶ Output of a layer used at a higher layer in the stack (besides serving as input to the next layer)
- ▶ Used to create **residual blocks**

Residual Block

- ▶ Let $f(\mathbf{x})$ be the function we aim to model
- ▶ With a residual block, we learn $g(\mathbf{x})$ and then compute $f(\mathbf{x}) = g(\mathbf{x}) + \mathbf{x}$

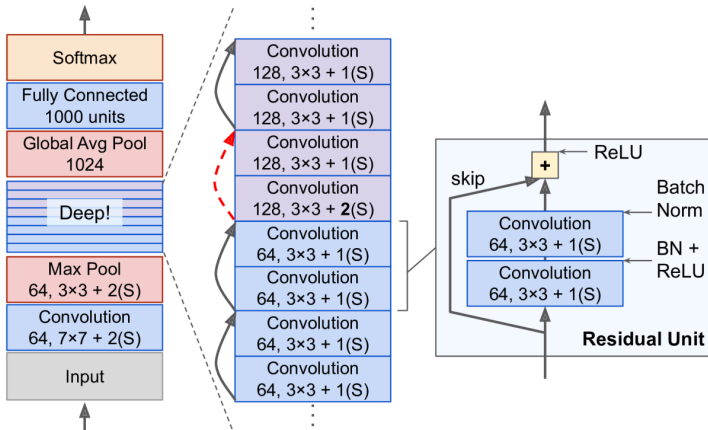


Residual Block: Why?

- ▶ The motivation behind residual learning is beyond the scope of the lecture
- ▶ Just an intuition: if weights are zero-initialized, your model will predict something close to 0
- ▶ With a residual block, you can easily initialize your model with the identity function (i.e., it just outputs the input)
- ▶ Being able to easily learn the identity function is beneficial for learning
- ▶ Furthermore, skip connections allow the NN to make progress even when many layers have not started learning yet

ResNet: Architecture

- ▶ Start and end similar to GoogLeNet
- ▶ Each res. block has 2 convolutional layers (and no pooling!)
- ▶ Batch Normalization and ReLU




Beyond ResNet

- ▶ ResNet is quite easy to modify
 - ▶ e.g., by tweaking number of channels and residual blocks, we can create different ResNet models, such as the deeper 152-layer ResNet-152
- ▶ Several other popular and successful CNN architectures proposed following ResNet
 - ▶ e.g., Google's Inception-v4 merged the ideas of GoogLeNet and ResNet
- ▶ Recently, proposals for **network design spaces** (e.g., RegNetX)
 - ▶ Given a few parameters, you automatically obtain a new NN model that enjoys some properties

Using Pre-Trained Models

- ▶ To use a well-known CNN model, you usually don't need to implement it
- ▶ Keras provides many **pre-trained** models with a single line of code

```
m = keras.applications.resnet50\  
    .ResNet50(weights="imagenet")
```

 tf_pretrained_cnn.ipynb

Transfer Learning

- ▶ What if you want to predict classes that were not considered in the existing model?
- ▶ **Transfer learning**: taking features learned on one problem, and leveraging them on a new, similar problem
- ▶ Especially useful for tasks where your dataset has too little data to train a full-scale model from scratch

Transfer Learning for DNNs

In the context of DL:

1. Take layers from a previously trained model
2. Freeze the lowest layers, so as to avoid destroying any of the information they contain during future training rounds
3. Add some new, trainable layers on top of the frozen layers
4. Train the new layers on your dataset
5. Optional - *Fine-tuning*: unfreeze the entire model and re-train it on the new data with a very low learning rate

Note: the term “fine-tuning” is often used to indicate the whole workflow, not only the last step

Example

- ▶ We load the pre-trained [Xception](#) CNN, a variant of GoogLeNet with improved performance
- ▶ The model has been trained to classify the ImageNet images (1,000 classes)
- ▶ We aim to reuse it for a binary classification problem (i.e., cat vs dog)
 - ▶ a “specialized” classification task

 tf_transfer.ipynb

References

- ▶ Goodfellow et al.: 9.1–9.3, 9.5
- ▶ D2L: 7, 8.1, 8.4, 8.6
- ▶ Hands-on ML: Chapter 14 (excluding object localization and detection)