Neural Networks 1 - Convolutional neural networks

18NES1 - Lecture 10, Summer semester 2024/25

Zuzana Petříčková

April 22th, 2025

What We Covered Last Time

Introduction to Convolutional Neural Networks

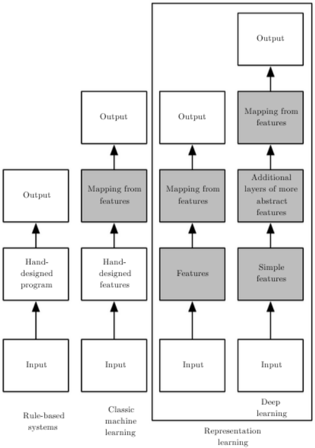
- Motivating example: bird species recognition
- The convolution operation – its meaning and parameters
- Convolutional neural network architecture (layers, filters, pooling)

This Week

Convolutional Neural Networks – Continued

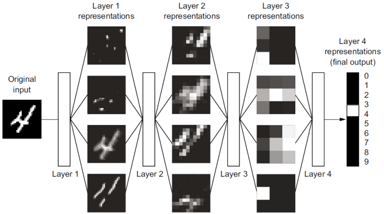
- Recap of fundamental concepts
- Classic CNN architecture + demonstration on MNIST data
- Visualizing how CNNs work
- Efficient processing of image data with data loaders
- Training a CNN from scratch
- Techniques to improve generalization in CNNs
- Introduction to transfer learning

Recap: Deep Learning



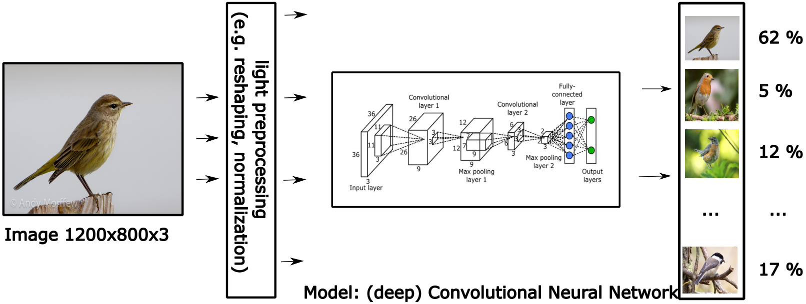
I. Goodfellow, Y. Bengio, A. Courville: Deep Learning, 2016, Figure 1.5

- Utilizes artificial neural networks with many layers (so-called deep networks)
- Models automatically learn to extract features from data – less manual preprocessing
- Architecture is often tailored to the specific data type (image, text, audio, ...)



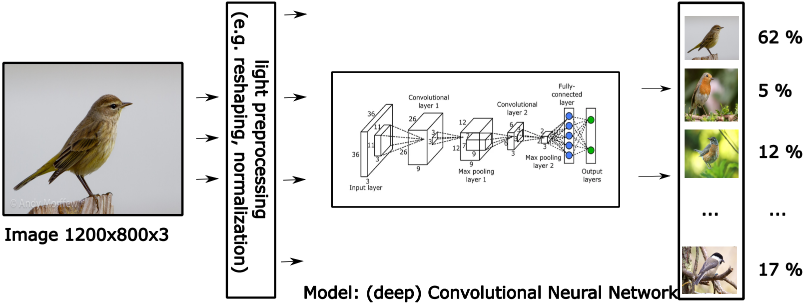
F. Chollet: Deep Learning with Python, Fig. 1.6

Convolutional Neural Network



- A specialized type of network for processing image data
- Efficient feature extraction using convolutional layers (filters)

Advantages of Convolutional Networks



- Preserve spatial relationships and local patterns in pixels
- Significantly fewer parameters compared to fully connected layers thanks to weight sharing
- Better scalability to large input images
- Robustness to translation and scale variations of objects

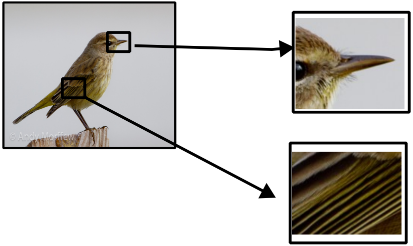
Convolutional Neural Network

- A neural network that includes convolutional layers

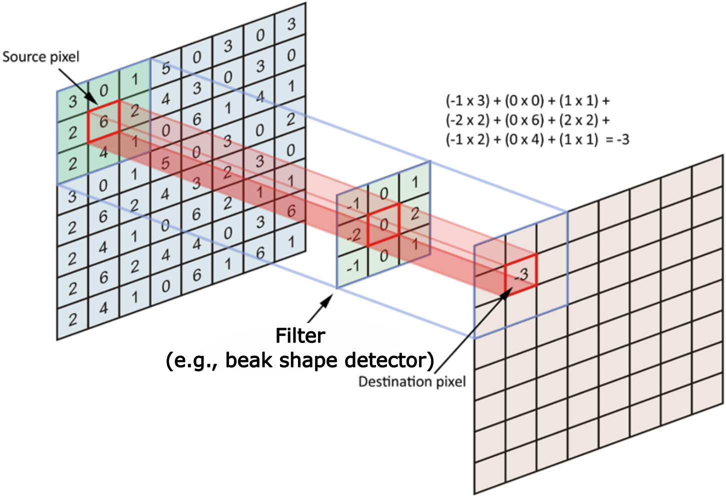
Convolutional Layer

- Consists of a set of filters (kernels, detectors)
- Each filter performs a convolution operation on the input image
- The result – a feature map – is passed to the next layer

Filter = detector of a particular pattern (feature) in the image



Recap: The Convolution Operation



Source: https://i.stack.imgur.com/YDusp.png

Convolution Operation

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Image 6x6 (black and white)

Convolutional Layer:

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1 (3x3)

-1	1	-1
-1	1	-1
-1	1	-1

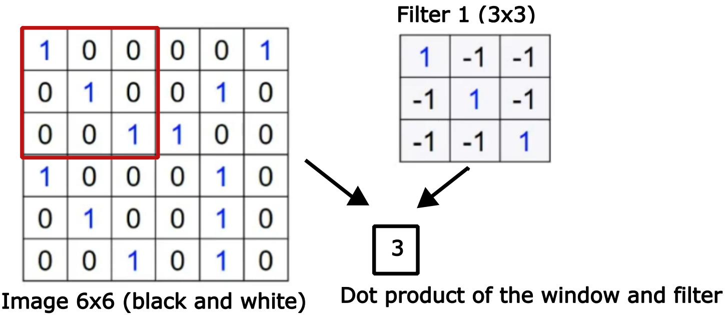
Filter 2 (3x3)

...

- Each convolutional layer contains several filters
- Each filter detects a pattern (feature) of size 3×3 pixels (e.g., diagonal edge, vertical edge, etc.)

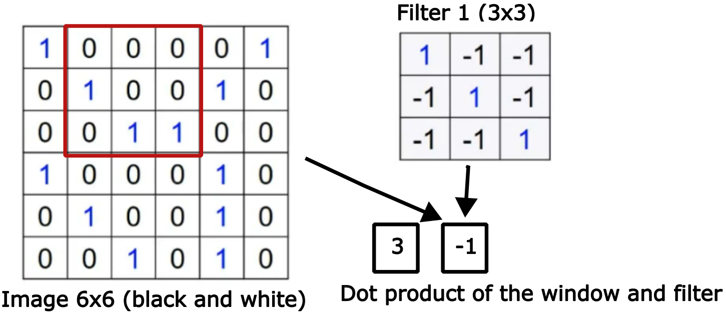
Example source: Petr Doležel – Convolutional Neural Network,
https://www.youtube.com/watch?v=-2vEi-Aa0FA

Convolution Operation



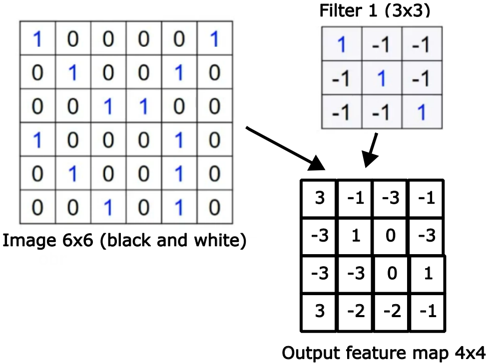
- We compute the dot product:
  $y = \sum_{i=1}^{9} w_i x_i + b$ (for flattened matrices)

Convolution Operation



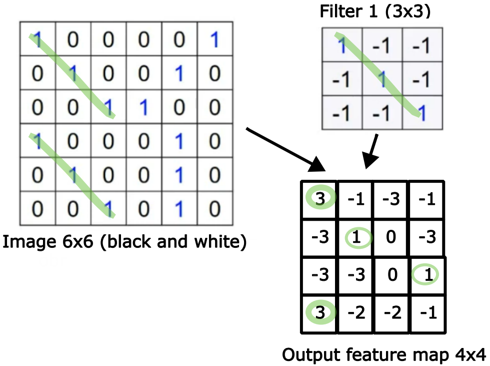
- Move the sliding window and compute another dot product

Convolution Operation



- By sliding the window over the image, we apply the filter to the entire input
- The result is a new 2×2 tensor – a **feature map**

Convolution Operation

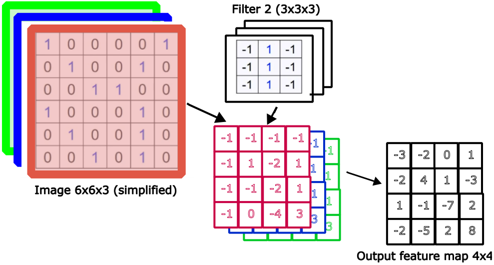


Feature Map

- Indicates where (and how strongly) the pattern represented by the filter appears in the input image
- Example: diagonal edge filter

Convolution Operation

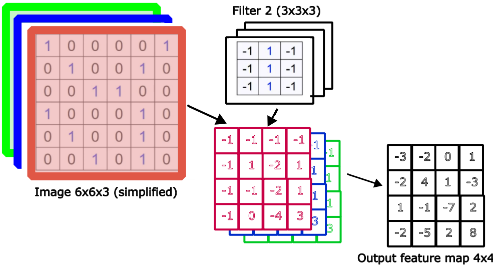
Color Image: 3 input channels – R, G, B



- Each filter has weights for **all input channels (R, G, B)**
- Computation: convolution is performed separately on each channel and the results are **summed**
- Each filter produces **one aggregated** output feature map
- The number of filters defines the **number of output channels**

Convolution Operation

Color Image: 3 input channels – R, G, B

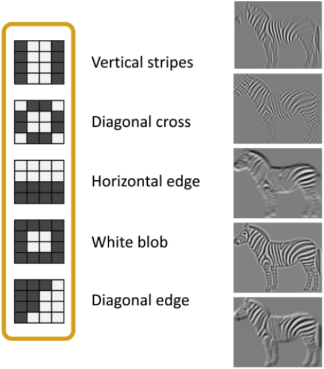


Weight Tensor in a Convolutional Layer:

- A 4-dimensional tensor with shape $u \times u \times c \times f$
  - $u \times u$ – spatial size of each filter (per channel)
  - c – number of input channels (e.g., 3 for RGB)
  - f – number of filters, i.e., number of output channels

Convolution Operation

Example: Zebra

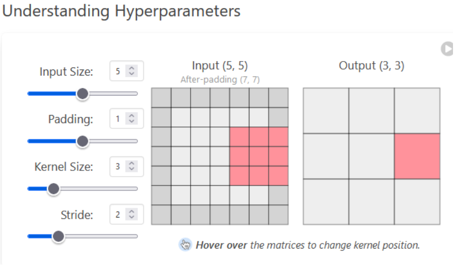


- Examples of other filters and their resulting feature maps

Convolution Operation

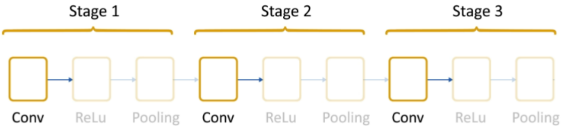
Convolution Operation Parameters

- Dimensions of the input image
- Padding – how borders are handled
- Filter size
- Stride – the step used to move the filter across the image



Great interactive visualization:
https://poloclub.github.io/cnn-explainer/

Classic CNN Architecture



Core idea: stack convolutional layers (or blocks) on top of each other

- The first convolutional layer detects simple features (e.g., edges, blobs)
- Each following layer extracts higher-level features

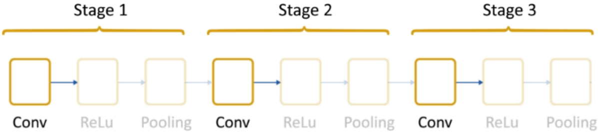
Hierarchical structure of features:

edges → shapes → object parts → whole objects

Classic CNN Architecture

Typical structure of a convolutional block:

- Convolutional layer
- Nonlinear activation function (e.g., ReLU)
- Pooling layer



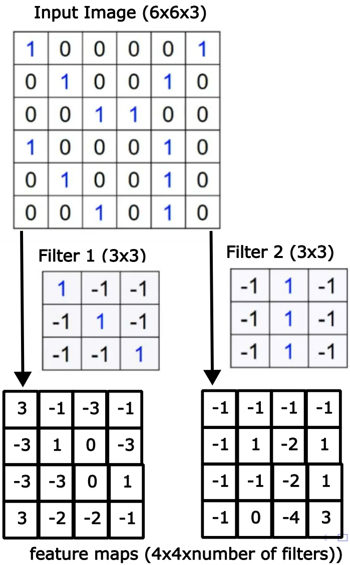
Pooling (Subsampling) Layer

- Reduces spatial resolution while preserving most of the relevant information
- A sliding window (e.g., 2×2) moves across the feature map, often with stride $= 2$
- Common operations: MAX (max pooling), AVERAGE (average pooling); no weights involved

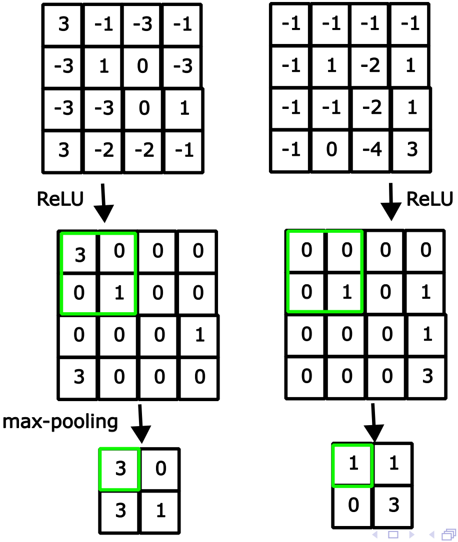
Why pooling?

- Condenses the information stored in the feature map
- Keeps track of where and how strongly a feature occurs
- Reduces the data size (e.g., $2 \times 2 \rightarrow 1$ value $= 75\%$ reduction)

Convolutional Block – Example: Convolutional Layer



Convolutional Block – Example: Pooling Layer



Convolutional Block

Why not just stack convolutional layers without pooling?

- The number of parameters grows with each added layer
- ,,Image" size stays (almost) the same, especially with "same" padding
  $\Rightarrow$ the size of the feature maps (and computation) keeps growing

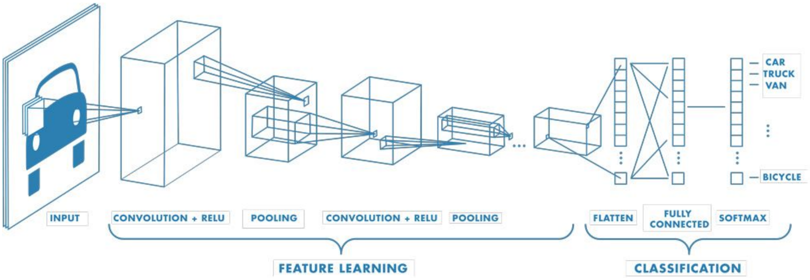
Pooling layer:

- Reduces data size while preserving information about feature presence and strength
- e.g., $2 \times 2 \rightarrow 1$ value = quarter size

Alternating convolution and pooling – bipyramidal effect:

- Spatial size decreases, number of feature maps increases

Classic CNN Architecture



Main components of a CNN:

- Convolutional blocks for feature extraction
- Flattening layer – converts the feature maps into a 1D vector
- Fully connected neural network for classification

Image source:

https://matlabacademy.mathworks.com/details/deep-learning-onramp/deeplearning

Bipyramidal Architecture

- One of the oldest architecture types: wide and shallow, with a deeper fully connected part – close to the basic layered schema

LeNet-5

- One of the original CNN architectures (Yann LeCun, 1998), relatively simple, trained on the MNIST dataset

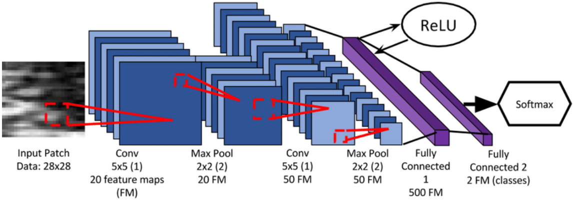


Image source: M. H. Yap et al., "Automated Breast Ultrasound Lesions Detection Using Convolutional Neural Networks," IEEE Journal of Biomedical and Health Informatics, vol. 22, 2018.

Bipyramidal Architecture

Typical structure of a bipyramidal architecture:

- The number of filters typically doubles in deeper layers (e.g., 32, 64, 128, ...)
- Most commonly used filter size: 3×3
- ReLu activation
- Max-pooling 2×2 is often paired with filter doubling
- When using several convolutional layers, we may not need many fully connected layers
- (Optionally) One or more fully connected layers are added for classification

visualize_cnn_mnist.ipynb

- Practical example: MNIST dataset (handwritten digits)

Representation of Input and Convolutional Layer Parameters

Input to the layer = 4D tensor of shape:
`(batch_size, height, width, channels)`

- For example, 8 RGB images of size 32×32 pixels:
  $\Rightarrow$ input tensor shape: `(8, 32, 32, 3)`
- In deeper layers, the input consists of feature maps from the previous layer:
  $\Rightarrow$ for example: `(8, 32, 32, 64)`

Weight tensor of a convolutional layer (filters) = 4D tensor of shape:
`(filter_height, filter_width, in_channels, out_channels)`

visualize_cnn_mnist.ipynb

Representation of Input and Convolutional Layer Parameters

Weight tensor of a convolutional layer (filters) = 4D tensor of shape:
`(filter_height, filter_width, in_channels, out_channels)`

- For example, 64 filters of size 3×3 for RGB input: `(3, 3, 3, 64)`
- Each filter "scans" the input image (or feature maps) and produces one output channel
- The result is a 4D output tensor: `(batch_size, new_height, new_width, out_channels)`

visualize_cnn_mnist.ipynb

Training a Convolutional Neural Network

- Typically trained using a variant of backpropagation (e.g., SGD)
- Mini-batch learning – the model requires a large amount of data
- A high number of trainable parameters

How to choose a suitable architecture in practice?

- We usually don't optimize the number of layers or neurons manually
- We pick a proven topology from the literature for the given type of task:
  - Bipyramidal architecture
  - One of the modern architectures (e.g., https://keras.io/api/applications/)

Examples: Visualization

visualize_cnn_mnist.ipynb

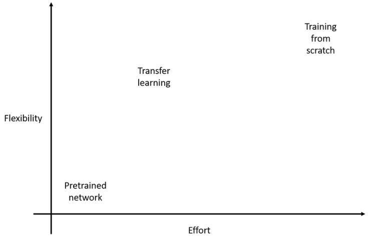
- Practical example: MNIST dataset (digits)
- Filter visualizations across layers, feature maps, and saliency visualization (pixel importance)

Useful links:

- Interactive CNN visualization: https://poloclub.github.io/cnn-explainer/
- MathWorks – activation visualization (face)

Ways to Build and Train a Convolutional Neural Network

- Training from scratch
- Using a pretrained model
- Transfer learning
- Fine-tuning a pretrained model



Source:

https://matlabacademy.mathworks.com/details/deep-learning-onramp/deeplearning

Example: Training a Model from Scratch on a Small Dataset (Flower Classification)

- Dataset: **Oxford 102 Flower Dataset** paperswithcode.com/dataset/oxford-102-flower
- Contains **8189 images** in **102 flower classes** (each class has 40–258 images)
- Dataset size: about **330 MB** (JPEG format)
- Suitable for testing CNN training from scratch and transfer learning

Task:

- For a start, we select a subset: 3 most frequent classes
- Split data into training, validation and test sets
- Train a basic bipyramidal CNN model on this data

CNN_from_scratch_flowers.ipynb

Example: Training a Model from Scratch on a Small Dataset (Flower Classification)

In this example, we demonstrate several practical techniques:

- Image preprocessing
- Efficient data loading using data loaders
- Using the Keras Functional API to build models
- Visualization of filters and feature maps for RGB images
- Data augmentation and regularization techniques

Example of a CNN Trained from Scratch on a Small Dataset: Flower Classification

Observations

- Compared to an MLP, the CNN learns relatively slowly.
- Compared to the model trained on MNIST, the model trained on Flowers creates more diverse filters — not only simple edges, but also more complex patterns.
- It's interesting to observe the saliency maps to see which parts of the image the model focused on for its predictions.
- The model showed signs of overfitting.

Extension: Classification into All 102 Classes
CNN_from_scratch_flowers_102.ipynb

- **Observation:** This model generalizes very poorly.
  → Could a regularization technique help here?

CNNs and Generalization

- CNNs often suffer from overfitting
- This is especially true when training data is limited (hundreds or a few thousand samples)
- Additionally, training can be quite slow
- How can we improve generalization?
  - **Standard regularization methods**
  - **Data augmentation**: increasing dataset diversity
  - **Transfer learning**

CNNs and Generalization

Common Regularization Techniques

- **Early stopping**
- **L1/L2 regularization** – typically used with ReLU units in convolutional and fully connected layers
- **Dropout** – adding a special dropout layer after each fully connected layer
- **Normalization** of inputs, weights, and layer outputs; a popular technique is Batch Normalization
- **Label smoothing** – adding noise to labels
- **Ensembling**

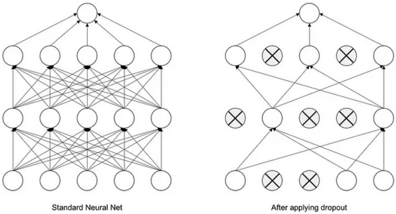
Especially relevant for CNNs:

- **Data augmentation**
- **Transfer learning**

CNN and Regularization

Dropout (Srivastava et al., 2014)

- Highly effective regularization technique
- Consists of randomly deactivating some hidden neurons during training
- During testing and inference, all neurons are active
- Implemented by inserting a special **dropout** layer after each fully connected layer

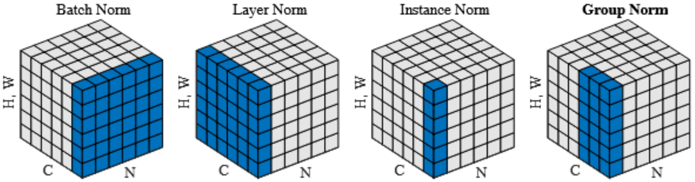


Srivastava, Nitish, et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, JMLR 2014

CNN and Regularization

Normalization of Layer Outputs

- Aims to fix the mean and variance of each layer's output
- Helps address the vanishing gradients problem
- Input to a convolutional layer is a 4D tensor of shape N $\times$ H $\times$ W $\times$ C
- N … batch size (number of samples), C … channels, H and W … spatial dimensions of the feature map
- Different normalization variants exist:

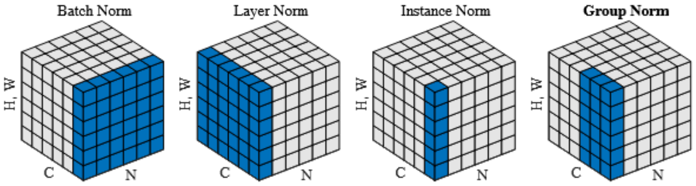


Wu, Y., et al. "Group Normalization", https://arxiv.org/pdf/1803.08494

CNN and Regularization

Normalization of Layer Inputs

- Implemented by adding an additional layer (e.g., after a convolutional layer)
- Results in faster training and reduced sensitivity to weight initialization
- Increases robustness to noise in the data (can replace Dropout for deep models)



Wu, Y., et al. "Group Normalization", https://arxiv.org/pdf/1803.08494

CNN and Regularization

Normalization: When It Fails

- Despite its theoretical advantages, normalization may **harm training** in practice:
  - **Small batch sizes and small datasets** lead to unstable or biased estimates of mean and variance.
  - May **slow down convergence** or **cause the model to get stuck in poor minima**.
  - **Interferes with Dropout, residual connections, or custom activations**.
- **Practical tip:**
  - On small or clean datasets, simpler models without normalization may perform better.

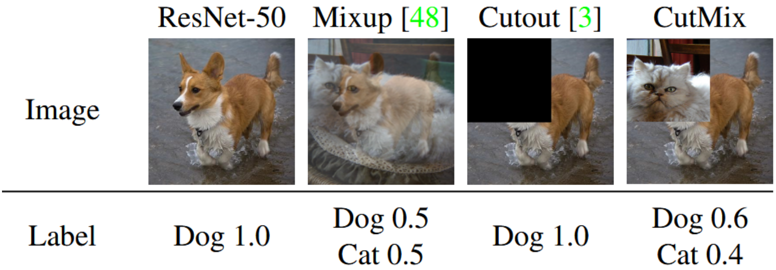
Data Augmentation

- Various (random) image transformations: rotation, shift, flip, skewing, resolution change, brightness and contrast adjustment, cropping, adding noise, blur, combinations
- In Keras, implemented via a dedicated layer

Source:

https://matlabacademy.mathworks.com/details/deep-learning-onramp/deeplearning

Data Augmentation – Popular Variants



Source: Yun et al., CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features, https://arxiv.org/pdf/1905.04899

Data Augmentation

Advantages:

- Artificially increases the size and diversity of the training dataset
- Helps prevent overfitting by exposing the model to a wider range of input variations
- Improves generalization to unseen data and robustness to distortions

Implementation in Keras:

- Easy to use via layers such as `RandomFlip`, `RandomRotation`, `RandomZoom`, etc.
- Augmentation happens **on the fly during training**, saving memory

Practical Examples of Regularization

regularization_cnn_mnist.ipynb

- Continuation of the MNIST example

CNN_from_scratch_flowers_3.ipynb, **CNN_from_scratch_flowers_102.ipynb**

- Continuation of the Flowers 102 example

Practical Examples of Regularization

Classification of Flowers into Three Classes – Observations

- The model learned the "simpler" task (3-class classification) quite well even without regularization, though it slightly overfit.
- Regularization (data augmentation, dropout, and early stopping) improved performance.
- Batch normalization did not improve results in this case (the model was relatively shallow).

Classification into All 102 Classes – Observations

- This time, the model without regularization generalized very poorly.
- Regularization helped, but only slightly.

Next Step

- How about using a pretrained model or transfer learning?

Using a Pretrained Model

What if we could use an existing model trained on a large dataset?

Pretrained models:

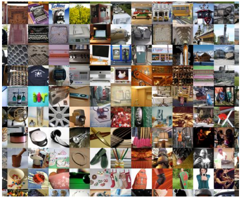
- https://keras.io/api/applications/
- Popular convolutional neural network architectures:
  - VGG16
  - MobileNet
  - ResNet
  - ...
- The model weights can be either randomly initialized or pretrained, typically on the **ImageNet** dataset

pretrained_model.ipynb

- A practical example of using a pretrained model

ImageNet Dataset

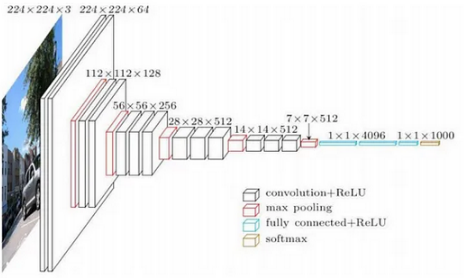
- 16 million color images from 20,000 categories
- Created as part of the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC, 2010–2017)
- This challenge sparked the breakthrough of convolutional neural networks in image recognition
- ImageNet became the standard benchmark dataset for model comparison (replacing MNIST)



Source: https://cs.stanford.edu/people/karpathy/cnnembed/cnn_embed_full_1k.jpg

Example of a Pretrained Model: VGGNet

- Karen Simonyan and Andrew Zisserman, 2014 – a family of models (e.g., VGG16, VGG19)
- Classic pyramidal architecture, relatively shallow (16 or 19 layers)



Source: https://medium.com/nerd-for-tech/vgg-16-easiest-explanation-12453b599526

Using a Pretrained Model

- Input images need to be resized to the expected dimensions and typically converted to RGB

- Required input size depends on the specific model (but is usually quite small, around 200 $\times$ 200)
- Images are typically rescaled and optionally cropped

From Pretrained Models to Transfer Learning

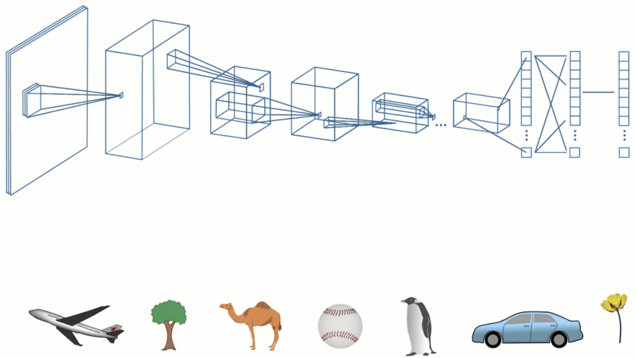
Using a pretrained model is a great starting point — but it usually won't be that simple.

- Although ImageNet contains 20,000 classes, the classification accuracy on your custom dataset may still be low.
- See our example: **pretrained_model.ipynb**

So how exactly do we adapt the model?

From a Pretrained Model to Transfer Learning

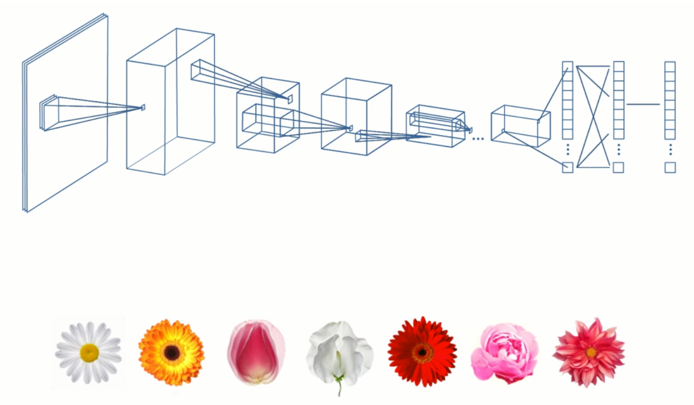
- Our pretrained model performs classification into 20,000 classes:



Source:
https://matlabacademy.mathworks.com/details/deep-learning-onramp/deeplearning

From a Pretrained Model to Transfer Learning

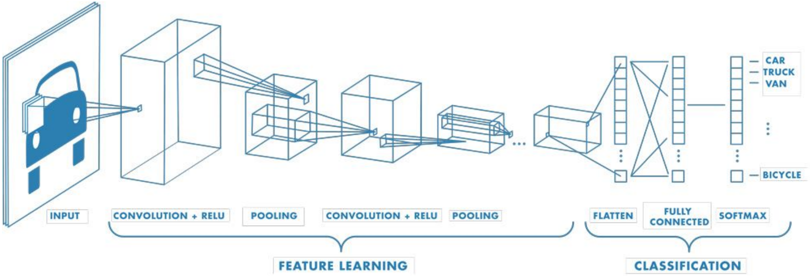
- But we need to classify into a different set of classes:



Source:

https://matlabacademy.mathworks.com/details/deep-learning-onramp/deeplearning

Recap: CNN Architecture



Components of a Convolutional Neural Network

- Convolutional base – extracts hierarchical features
- Flattening layer – converts data to a numeric vector
- Fully connected neural network for classification – **classification head**

Source:
https://matlabacademy.mathworks.com/details/deep-learning-onramp/deeplearning

From Pretrained Model to Transfer Learning

How exactly? First idea:

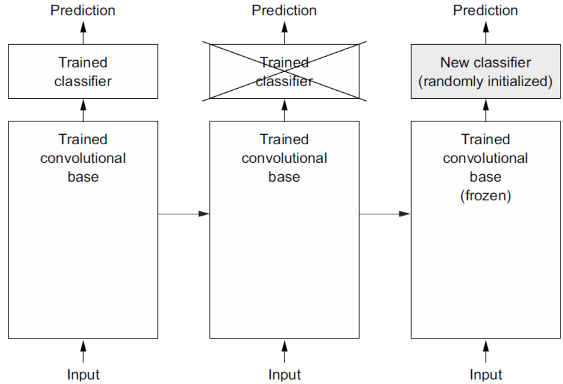
- Take a network pretrained on ImageNet
- Remove its classification head
- Use it to extract features from the new data → create a new training set
- Build a new fully connected neural network and train it on the extracted features

This approach is efficient, but often impractical:

- It's static – hard to apply built-in data augmentation tools

Transfer Learning

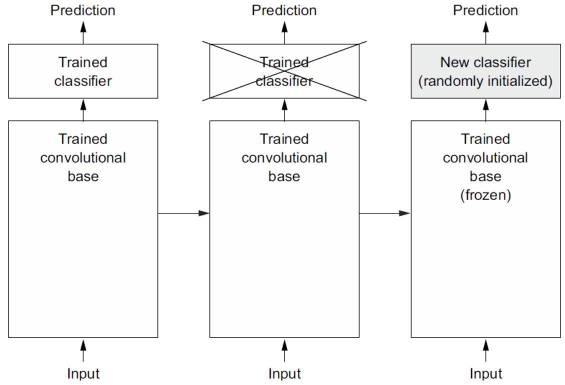
- Take a network pretrained on ImageNet
- Replace its classification head (or just its top part) with a new one (randomly initialized)



F. Chollet: Deep Learning with Python, Fig. 8.2

Transfer Learning

- Train the new classification head on your new data (keep the earlier layers frozen)



F. Chollet: Deep Learning with Python, Fig. 8.2

Transfer Learning – Practical Notes

- Typically use a smaller learning rate than when training from scratch
- Regularization (dropout, data augmentation) is useful

Examples:
CNN_transfer_learning_flowers_3.ipynb,
CNN_transfer_learning_flowers_102.ipynb

- Continuation of the Flowers 102 example