Module 3

Computer Vision using Convolutional Neural Networks

Introduction

It wasn’t until recently that computers were able to reliably perform seemingly trivial tasks such as detecting a puppy in a picture or recognizing spoken words.

Why are these tasks so effortless to us humans?

The answer lies in the fact that perception largely takes place outside the realm of our consciousness, within specialized visual, auditory, and other sensory modules in our brains.
By the time sensory information reaches our consciousness, it is already adorned with high-level features; for example, when you look at a picture of a cute puppy, you cannot choose not to see the puppy, not to notice its cuteness. Nor can you explain how you recognize a cute puppy; it’s just obvious to you.

Introduction

Convolutional neural networks (CNNs) have been used in computer image recognition since the 1980s. Over the last 10 years, thanks to the increase in computational power, the amount of available training data, and the tricks presented earlier for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks.

They power image search services, self-driving cars, automatic video classification systems, and more.

CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing.

Introduction

An important milestone was a 1998 paper by Yann LeCun et al. that introduced the famous LeNet-5 architecture, which was widely used by banks to recognize handwritten digits on checks.

This architecture has fully connected layers and introduces two new building blocks: convolutional layers and pooling layers.

Introduction

Why not simply use a deep neural network with fully connected layers for image recognition tasks?

Introduction

Why not simply use a deep neural network with fully connected layers for image recognition tasks?

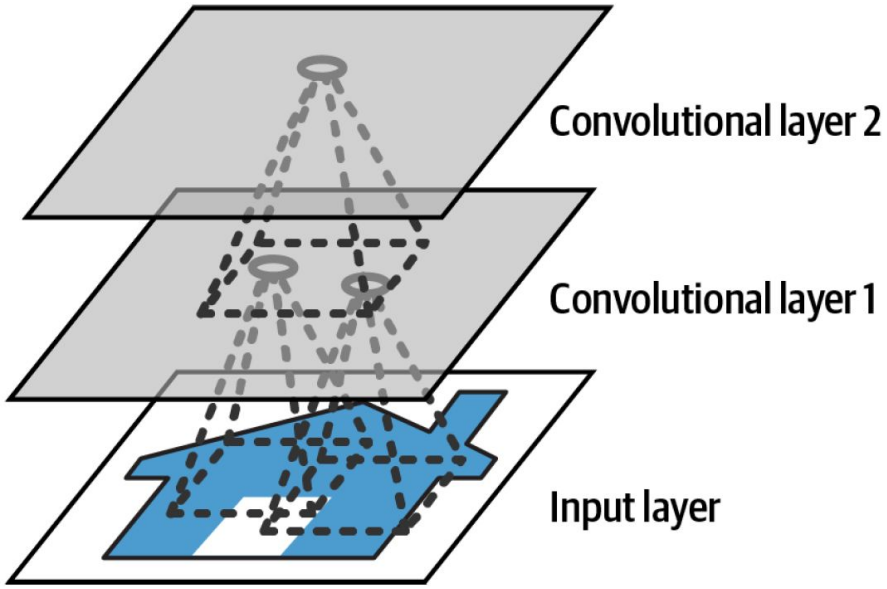
Unfortunately, although this works fine for small images (e.g., MNIST), it breaks down for larger images because of the huge number of parameters it requires. For example, a 100 x 100 pixel image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer.

Convolutional Layers

The most important building block of a CNN is the convolutional layer: neurons in the first convolutional layer are not connected to every single pixel in the input image, but only to pixels in their receptive fields.

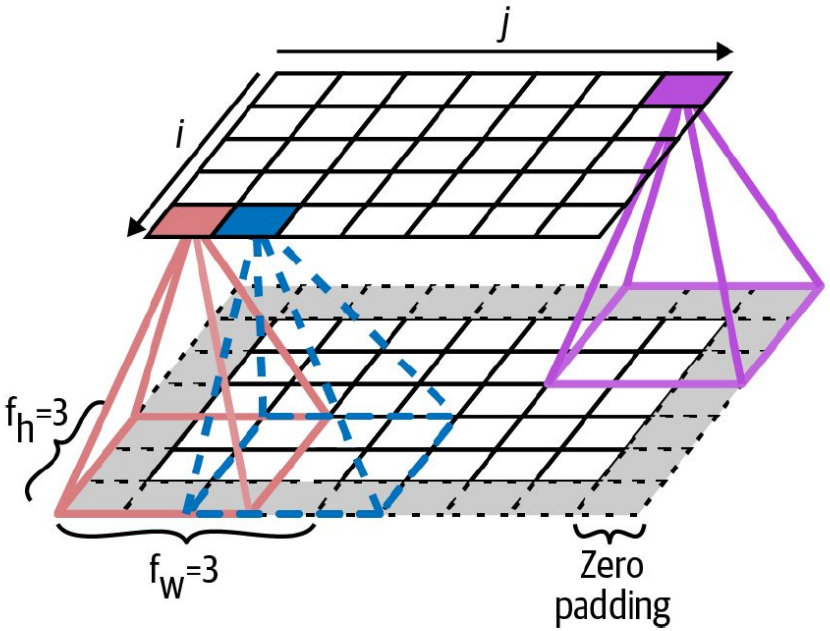
Each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer.

This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on.



Convolutional Layers

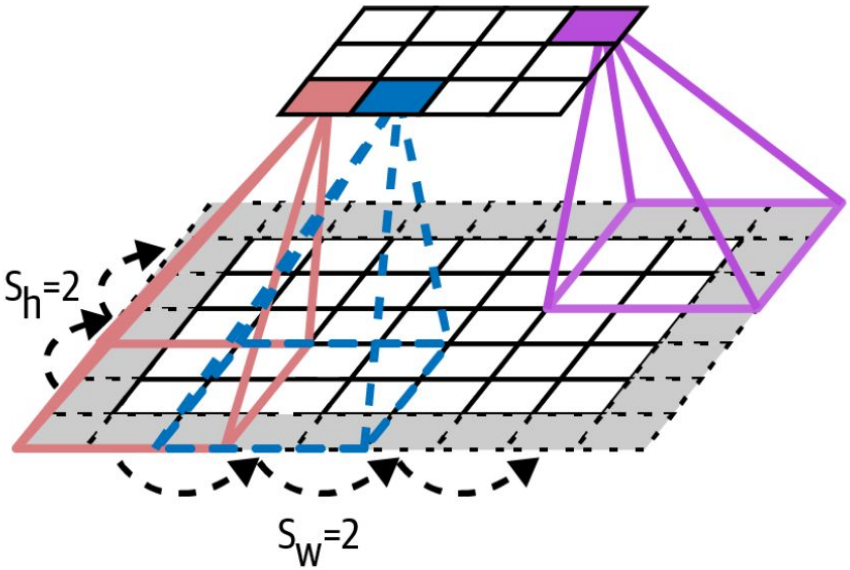
In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs.
This is called zero padding.



Convolutional Layers

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields.

This dramatically reduces the model's computational complexity. The horizontal or vertical step size from one receptive field to the next is called the stride.

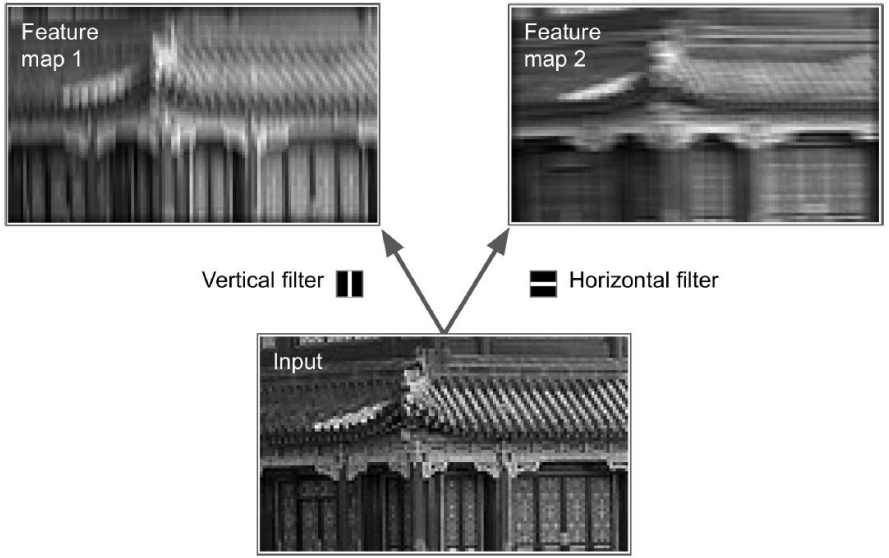


Filters

A neuron's weights can be represented as a small image. Figure shows two possible sets of weights, called filters (or convolution kernels, or just kernels).

The first one is represented as a black square with a vertical white line in the middle; neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will be multiplied by 0, except for the ones in the central vertical line).

The second filter is a black square with a horizontal white line in the middle. Neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

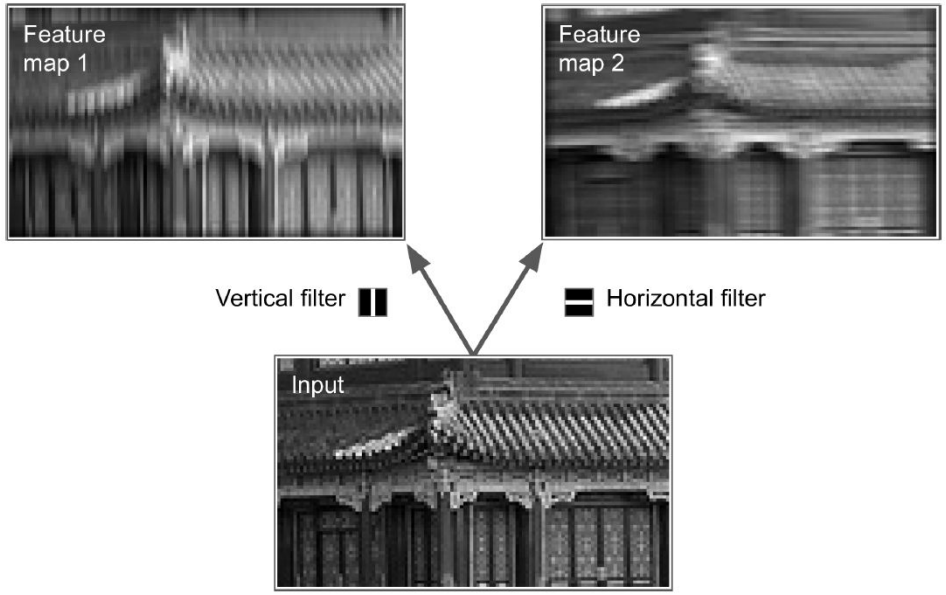


Filters

Notice that in Feature map 1, vertical white lines get enhanced while the rest gets blurred.

Similarly, the upper-right image is what you get if all neurons use the same horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out.

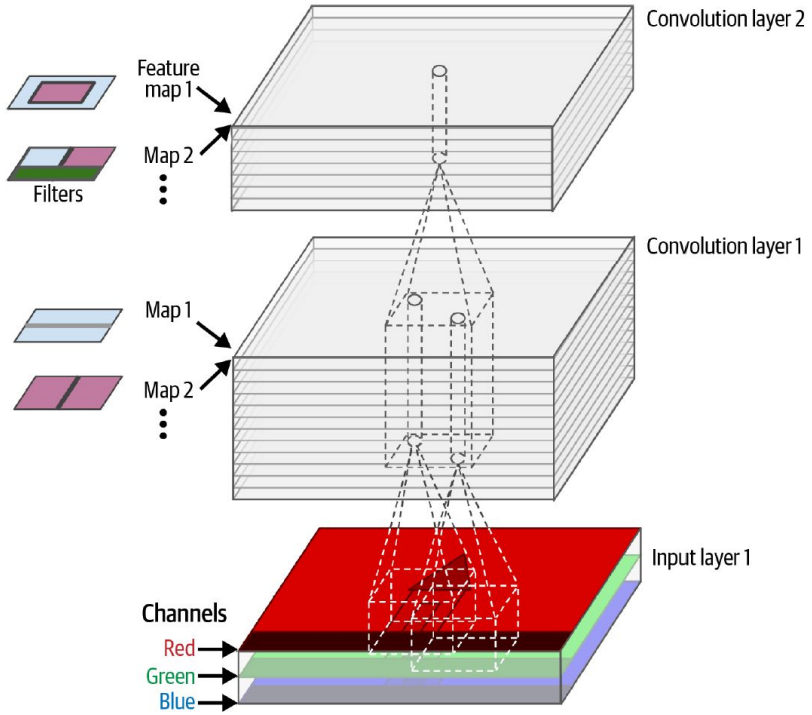
You won't have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task.



Stacking Multiple Feature Maps

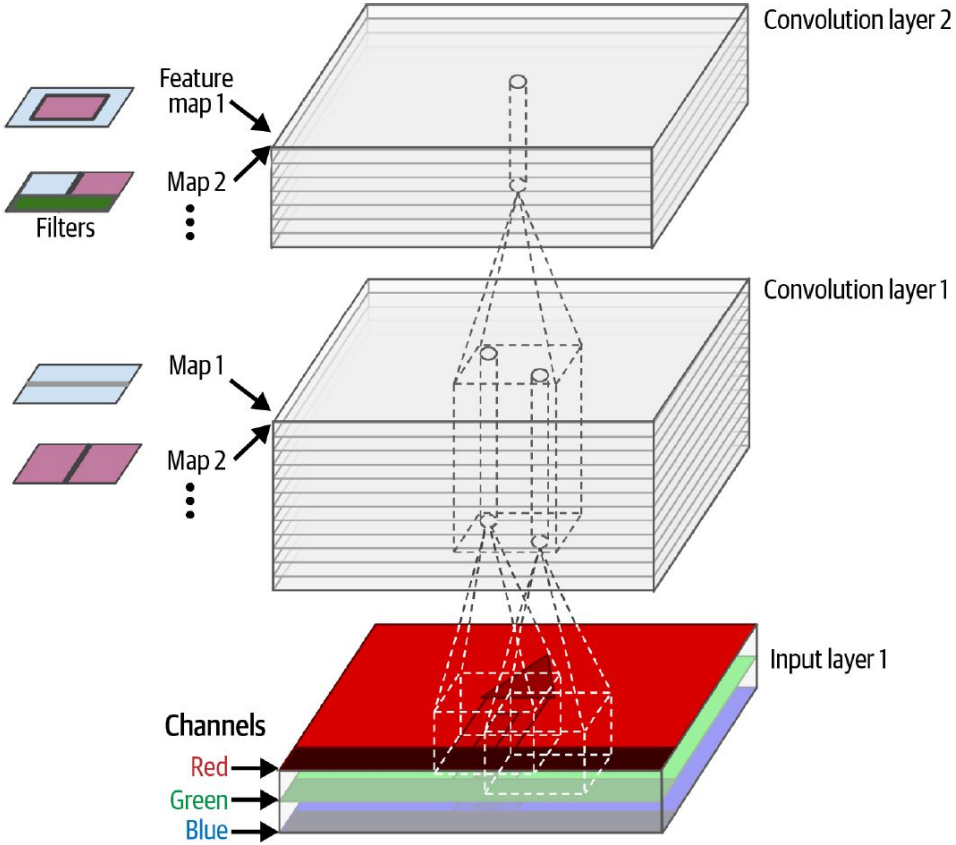
A convolutional layer can have multiple filters (you decide how many) and outputs one feature map per filter.

The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model.
Once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location.

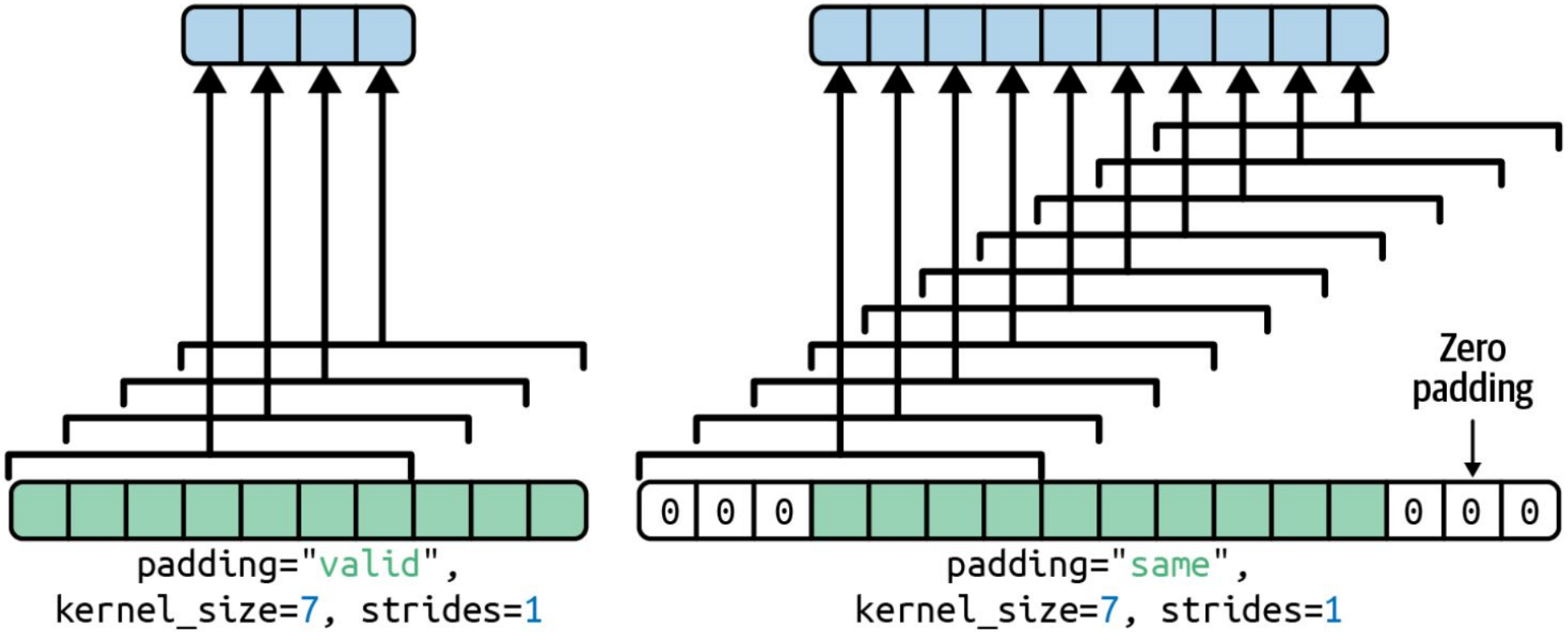


Stacking Multiple Feature Maps

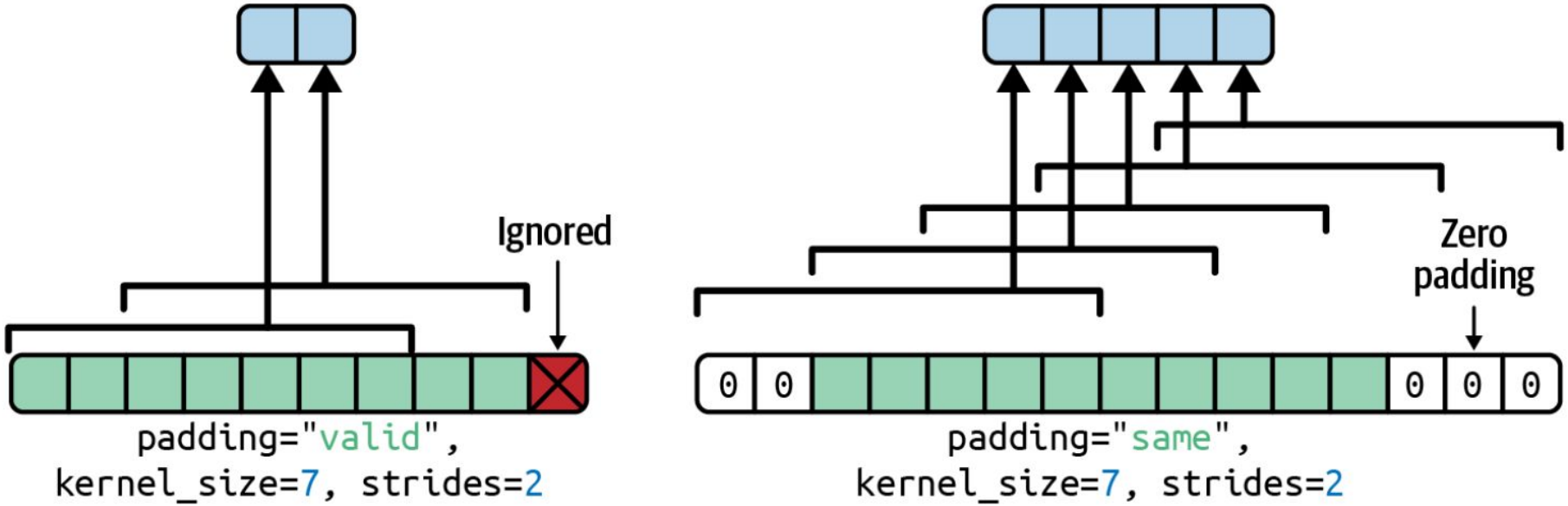
Input images are composed of multiple sublayers: one per color channel. There are typically three: red, green, and blue (RGB).
Grayscale images have just one channel, but some images may have many more - for example, satellite images that capture extra light frequencies (such as infrared).



Convolutional Layers

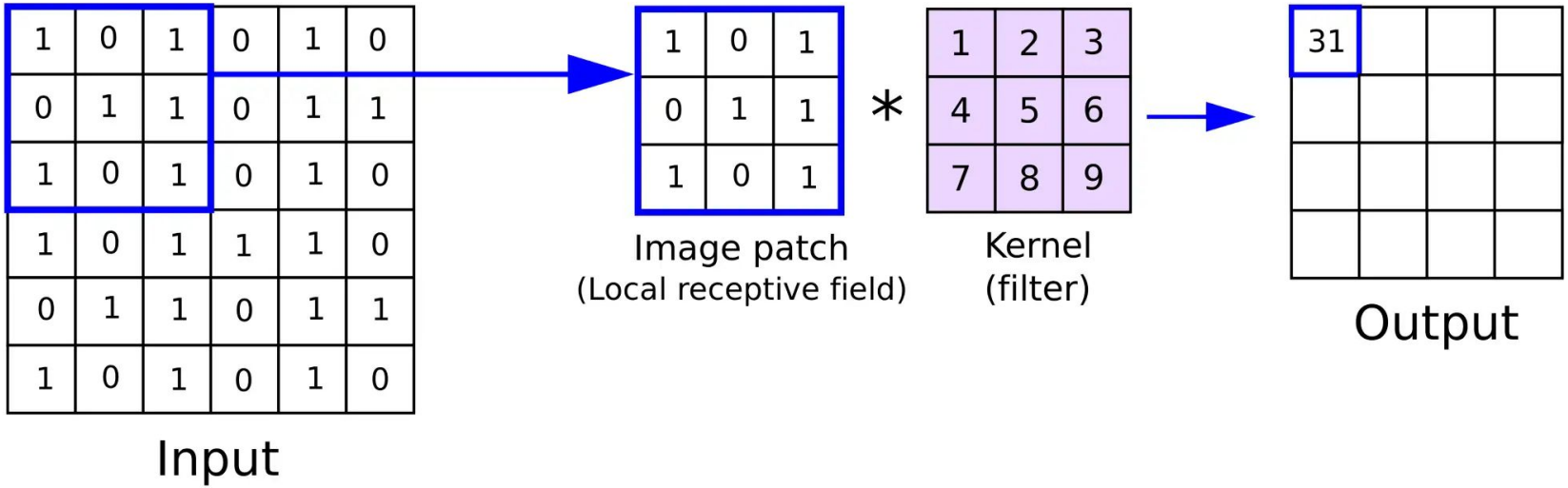


Convolutional Layers



With strides greater than 1, the output is much smaller even when using "same" padding (and "valid" padding may ignore some inputs)

Convolutional Layers



https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

Memory Requirements

Challenge with CNNs is that the convolutional layers require a huge amount of RAM. This is especially true during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass.

Memory Requirements

For example, consider a convolutional layer with 200 5 × 5 filters, with stride 1 and "same" padding. If the input is a 150 × 100 RGB image (three channels), then the number of parameters is (5 × 5 × 3 + 1) × 200 = 15,200 (the + 1 corresponds to the bias terms).

Each of the 200 feature maps contains 150 × 100 neurons, and each of these neurons needs to compute a weighted sum of its 5 × 5 × 3 = 75 inputs: that’s a total of 225 million float multiplications.

Not as bad as a fully connected layer, but still quite computationally intensive.

If the feature maps are represented using 32-bit floats, then the convolution layer output will occupy 200 × 150 × 100 × 32 = 96 million bits (12 MB) of RAM. And that’s just for one instance - if a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM!

Memory Requirements

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.

If training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, removing a few layers, using 16-bit floats instead of 32-bit floats, or distributing the CNN across multiple devices.

Pooling Layers

Goal is to subsample (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting).

You must define its size, the stride, and the padding type, just like before.

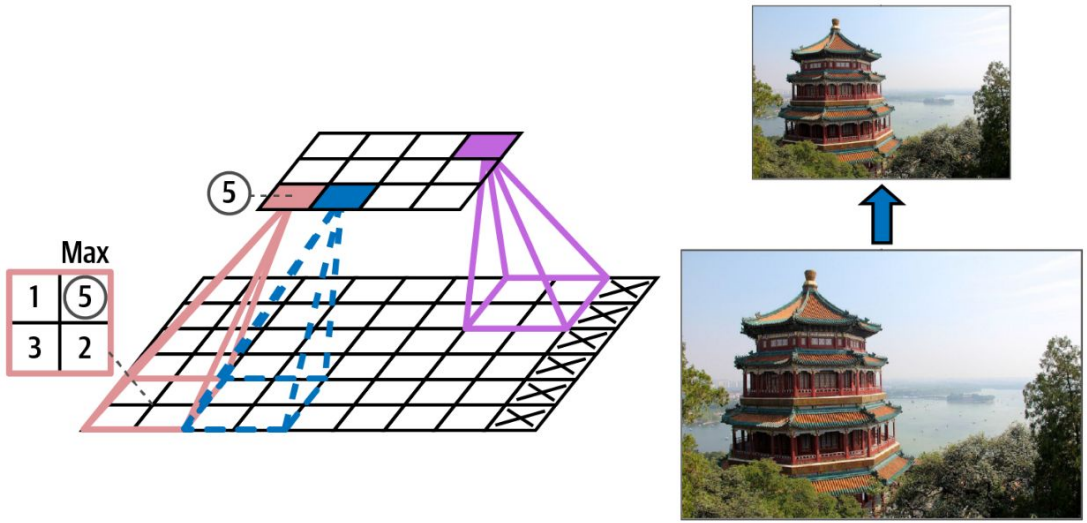
A pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean.

Max-Pooling

A 2 × 2 pooling kernel, with a stride of 2 and no padding.

Only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped.

Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).



Pooling Layers

Max pooling has some downside too.

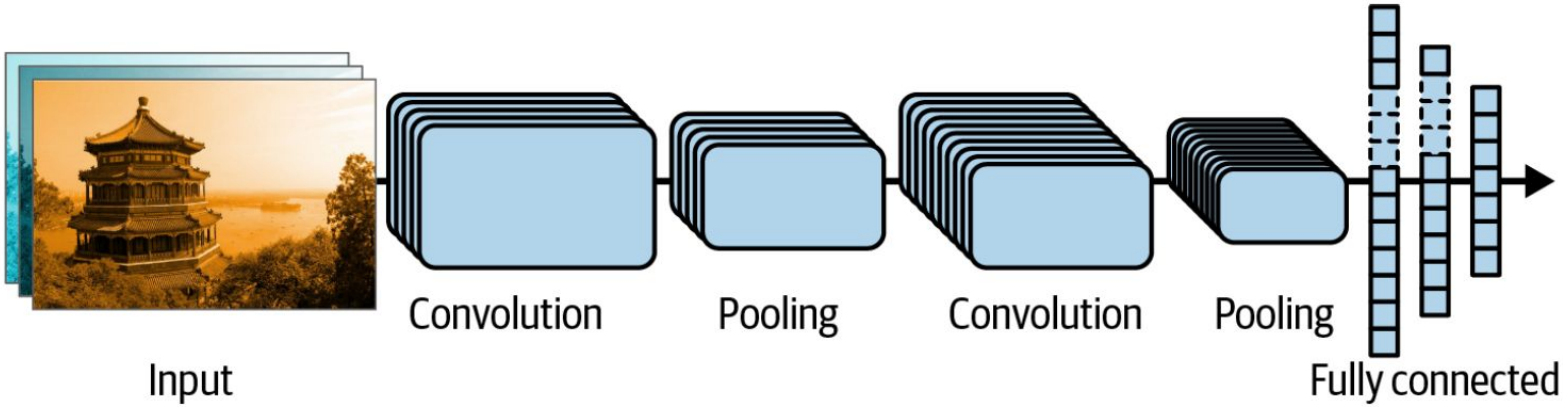
It is very destructive: even with a tiny 2 × 2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values.

CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on.

The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps).

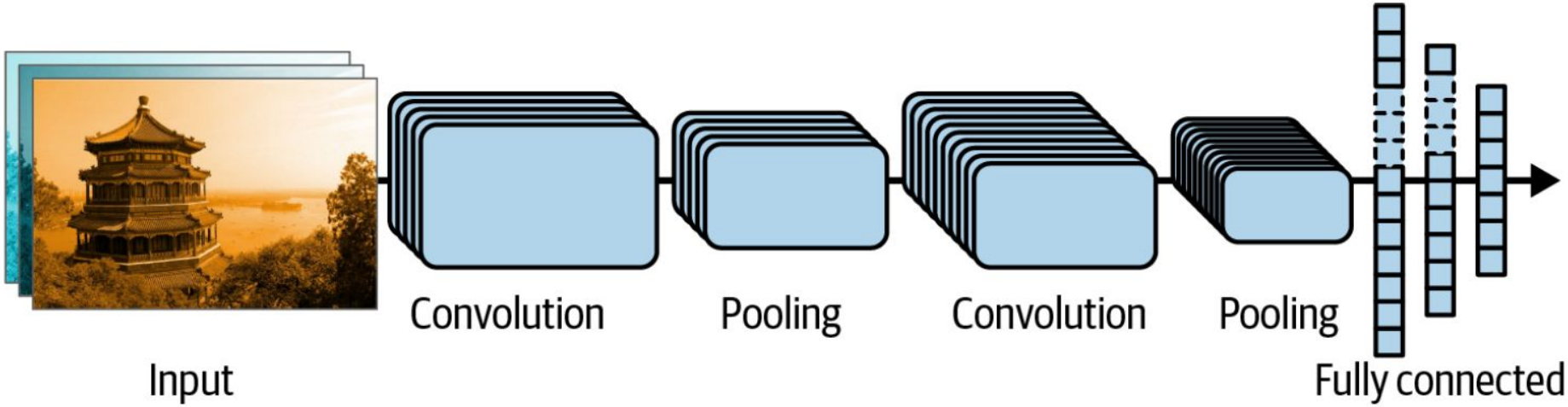
At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).



CNN Architectures

A common mistake is to use convolution kernels that are too large. For example, instead of using a convolutional layer with a 5 × 5 kernel, stack two layers with 3 × 3 kernels: it will use fewer parameters and require fewer computations, and it will usually perform better.

One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5 × 5), usually with a stride of 2 or more. This will reduce the spatial dimension of the image without losing too much information.



CNN Architectures

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field.

A good measure of this progress is the error rate in competitions such as the ILSVRC ImageNet challenge. In this competition, the top-five error rate for image classification - that is, the number of test images for which the system's top five predictions did *not* include the correct answer - fell from over 26% to less than 2.3% in just six years.

The images are fairly large (e.g., 256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds). Looking at the evolution of the winning entries is a good way to understand how CNNs work, and how research in deep learning progresses.

LeNet-5

The LeNet-5 architecture is the most widely known CNN architecture.

It was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST)

A stack of convolutional layers and pooling layers, followed by a dense network. Perhaps the main difference with more modern classification CNNs is the activation functions: today, we would use ReLU instead of tanh and softmax instead of RBF.

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

AlexNet

The AlexNet CNN architecture won the 2012 ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best competitor achieved only 26%!

AlexNet was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton.

It is similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of one another, instead of stacking a pooling layer on top of each convolutional layer.

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6×6	3×3	2	`valid`	–
C7	Convolution	256	13×13	3×3	1	`same`	ReLU
C6	Convolution	384	13×13	3×3	1	`same`	ReLU
C5	Convolution	384	13×13	3×3	1	`same`	ReLU
S4	Max pooling	256	13×13	3×3	2	`valid`	–
C3	Convolution	256	27×27	5×5	1	`same`	ReLU
S2	Max pooling	96	27×27	3×3	2	`valid`	–
C1	Convolution	96	55×55	11×11	4	`valid`	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

AlexNet

To reduce overfitting, the authors used two regularization techniques.

1. Applied dropout with a 50% dropout rate during training to the outputs of layers F9 and F10.
2. Performed data augmentation by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6×6	3×3	2	`valid`	–
C7	Convolution	256	13×13	3×3	1	`same`	ReLU
C6	Convolution	384	13×13	3×3	1	`same`	ReLU
C5	Convolution	384	13×13	3×3	1	`same`	ReLU
S4	Max pooling	256	13×13	3×3	2	`valid`	–
C3	Convolution	256	27×27	5×5	1	`same`	ReLU
S2	Max pooling	96	27×27	3×3	2	`valid`	–
C1	Convolution	96	55×55	11×11	4	`valid`	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

ZFNet

A variant of AlexNet called *ZFNet* was developed by Matthew Zeiler and Rob Fergus and won the 2013 ILSVRC challenge.

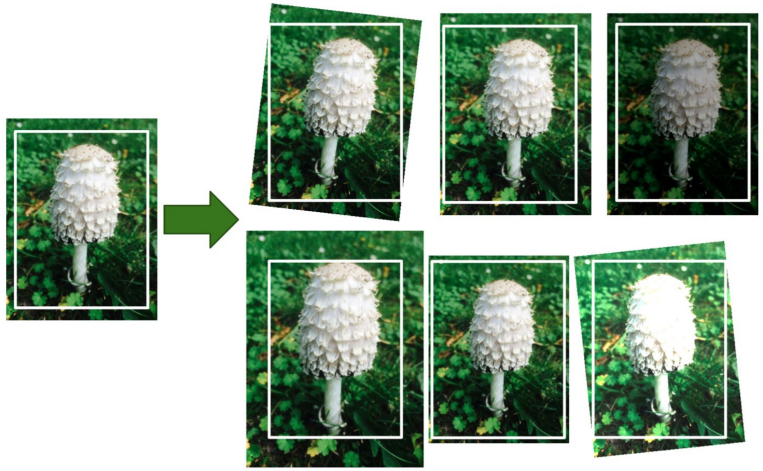
It is essentially AlexNet with a few tweaked hyperparameters (number of feature maps, kernel size, stride, etc.).

Data Augmentation

Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance.

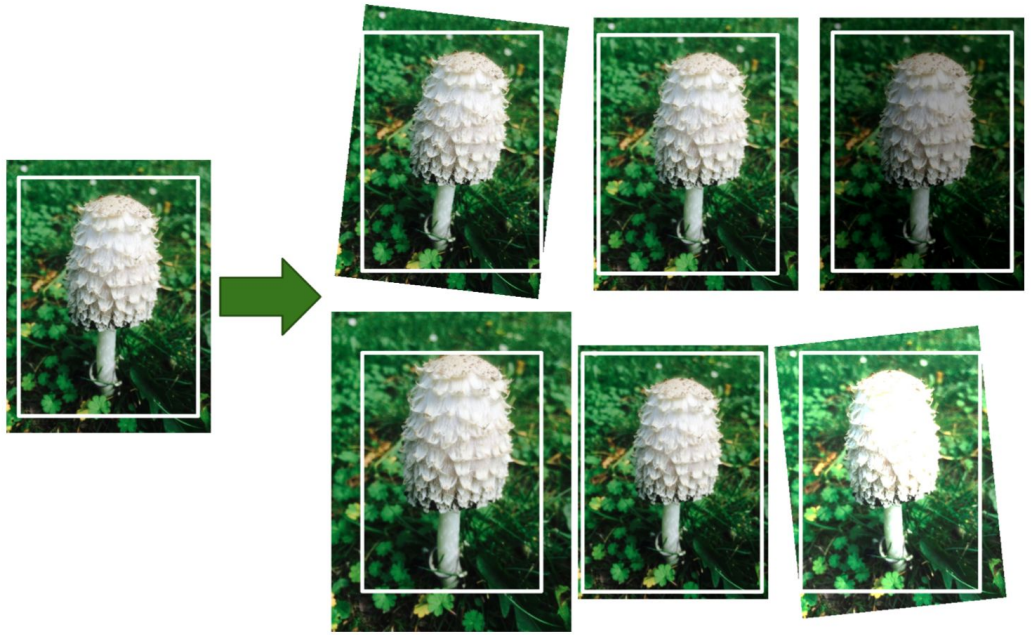
This reduces overfitting, making this a regularization technique.

The generated instances should be as realistic as possible: ideally, given an image from the augmented training set, a human should not be able to tell whether it was augmented or not.



Data Augmentation

You can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set.



Data augmentation is also useful when you have an unbalanced dataset: you can use it to generate more samples of the less frequent classes. This is called the *synthetic minority oversampling technique*, or SMOTE for short.

GoogleNet

The GoogLeNet architecture was developed by Christian Szegedy et al. from Google Research,and it won the ILSVRC 2014 challenge by pushing the top-five error rate below 7%.

This great performance came in large part from the fact that the network was much deeper than previous CNNs. This was made possible by subnetworks called *inception modules*, which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

GoogleNet

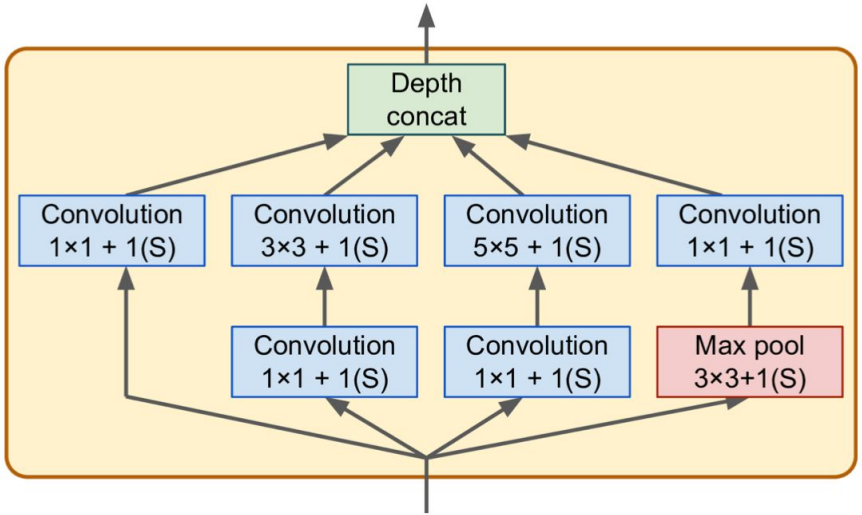
The notation “3 × 3 + 1(S)” means that the layer uses a 3 × 3 kernel, stride 1, and "same" padding.

The input signal is first fed to four different layers in parallel. All convolutional layers use the ReLU activation function.

Top convolutional layers use different kernel sizes (1 × 1, 3 × 3, and 5 × 5), allowing them to capture patterns at different scales.

Every single layer uses a stride of 1 and "same" padding (even the max pooling layer), so their outputs all have the same height and width as their inputs. Makes it possible to concatenate all the outputs along the depth dimension in the final *depth concatenation layer* (i.e., to stack the feature maps from all four top convolutional layers).

Architecture of an inception module

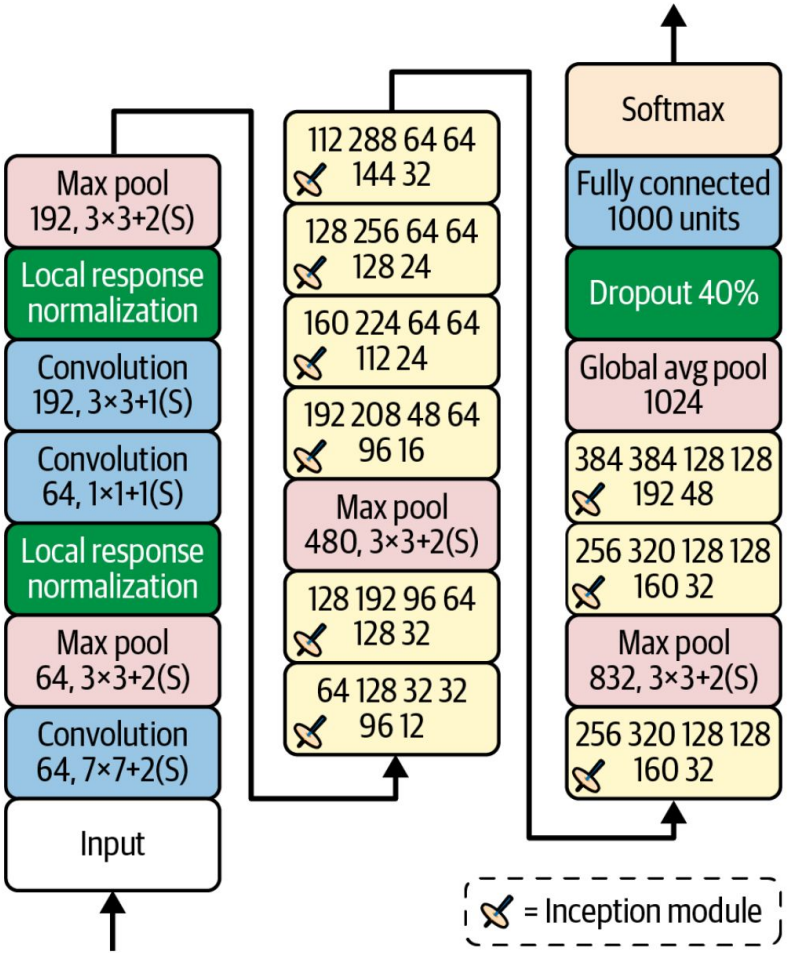


GoogleNet

Let's look at the architecture of the GoogLeNet CNN.

The architecture is so deep that it has to be represented in three columns, but GoogLeNet is actually one tall stack, including nine inception modules (the boxes with the spinning tops). The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module.

All the convolutional layers use the ReLU activation function.



GoogleNet

Several variants of the GoogLeNet architecture were later proposed by Google researchers, including Inception-v3 and Inception-v4, using slightly different inception modules to reach even better performance.

VGGNet

The runner-up in the ILSVRC 2014 challenge was VGGNet

Karen Simonyan and Andrew Zisserman, from the Visual Geometry Group (VGG) research lab at Oxford University, developed a very simple and classical architecture;

it had 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used small 3 × 3 filters, but it had many of them.

ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a Residual Network (ResNet), that delivered an astounding top-five error rate under 3.6%.

The winning variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers).

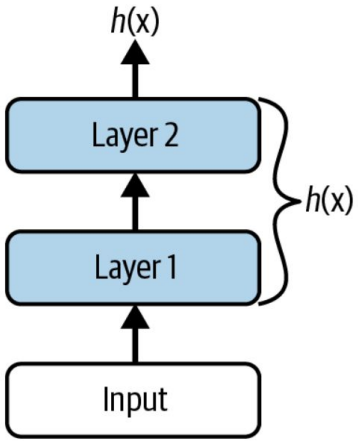
It confirmed the general trend: computer vision models were getting deeper and deeper, with fewer and fewer parameters.

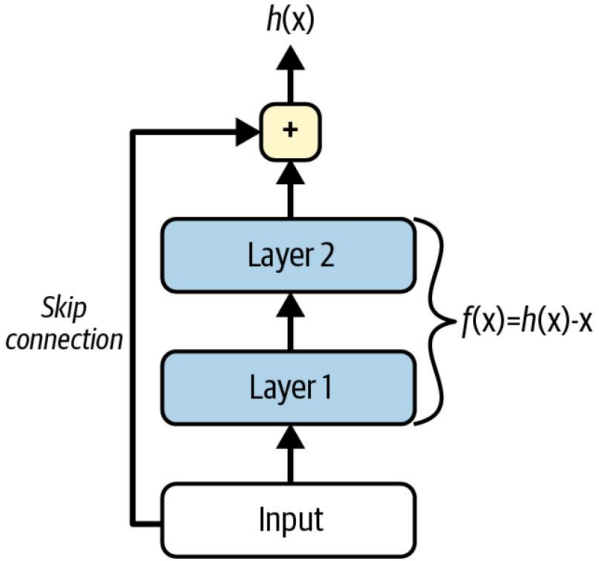
The key to being able to train such a deep network is to use *skip connections* (also called *shortcut connections*): the signal feeding into a layer is also added to the output of a layer located higher up the stack.

ResNet

When training a neural network, the goal is to make it model a target function $h(\mathbf{x})$.

If you add the input **x** to the output of the network (i.e., you add a skip connection), then the network will be forced to model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ rather than $h(\mathbf{x})$. This is called *residual learning*.

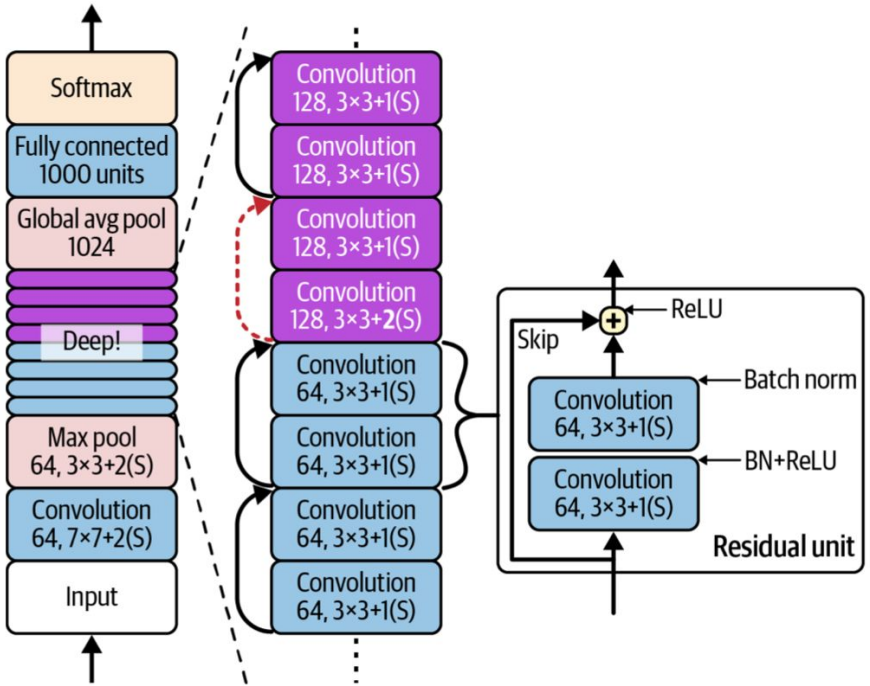




The deep residual network can be seen as a stack of *residual units* (RUs), where each residual unit is a small neural network with a skip connection.

ResNet Architecture

It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of residual units. Each residual unit is composed of two convolutional layers (and no pooling layer!), with batch normalization (BN) and ReLU activation, using 3 × 3 kernels and preserving spatial dimensions (stride 1, "same" padding).



ResNet

Different variations of the architecture exist, with different numbers of layers.

ResNet-34 is a ResNet with 34 layers (only counting the convolutional layers and the fully connected layer)

Google’s Inception-v4 architecture merged the ideas of GoogLeNet and ResNet and achieved a top-five error rate of close to 3% on ImageNet classification.

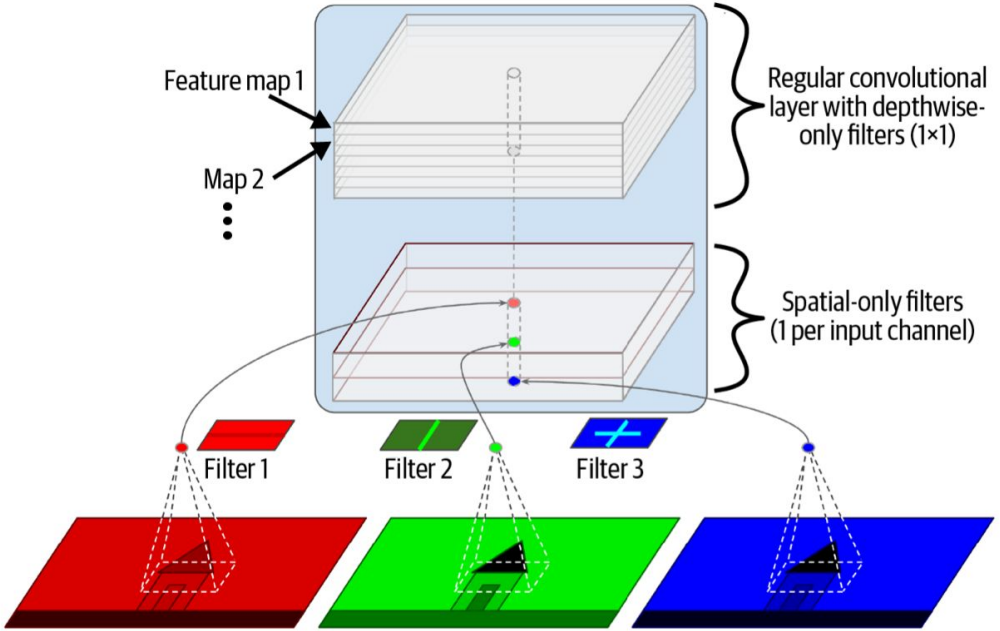
Xception

Another variant of the GoogLeNet architecture: Xception (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the author of Keras), and it significantly outperformed Inception-v3 on a huge vision task (350 million images and 17,000 classes).

Just like Inception-v4, it merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a *depthwise separable convolution layer*. These layers had been used before in some CNN architectures, but they were not as central as in the Xception architecture.

Xception

A separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately. Thus, it is composed of two parts: the first part applies a single spatial filter to each input feature map, then the second part looks exclusively for cross-channel patterns - it is just a regular convolutional layer with 1 × 1 filters.



Xception

Since separable convolutional layers only have one spatial filter per input channel, you should avoid using them after layers that have too few channels, such as the input layer.

For this reason, the Xception architecture starts with 2 regular convolutional layers, but then the rest of the architecture uses only separable convolutions (34 in all), plus a few max pooling layers and the usual final layers (a global average pooling layer and a dense output layer).

Xception

Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and they often perform better. Consider using them by default, except after layers with few channels (such as the input channel). In Keras, just use SeparableConv2D instead of Conv2D. Keras also offers a DepthwiseConv2D layer that implements the first part of a depthwise separable convolutional layer (i.e., applying one spatial filter per input feature map).

MobileNet

MobileNets are streamlined models designed to be lightweight and fast, making them popular in mobile and web applications.

They are based on depthwise separable convolutional layers, like Xception.

The authors proposed several variants, trading a bit of accuracy for faster and smaller models.

EfficientNet

Proposed a method to scale any CNN efficiently, by jointly increasing the depth (number of layers), width (number of filters per layer), and resolution (size of the input image) in a principled way. This is called *compound scaling*.

They used neural architecture search to find a good architecture for a scaled-down version of ImageNet (with smaller and fewer images), and then used compound scaling to create larger and larger versions of this architecture.

When EfficientNet models came out, they vastly outperformed all existing models, across all compute budgets, and they remain among the best models out there today.

Choosing the Right CNN Architecture

With so many CNN architectures, how do you choose which one is best for your project?

Well, it depends on what matters most to you: Accuracy? Model size (e.g., for deployment to a mobile device)? Inference speed on CPU? On GPU?

Choosing the Right CNN Architecture

For each model, the table shows the Keras class name to use, the model's size in MB, the top-1 and top-5 validation accuracy on the ImageNet dataset, the number of parameters (millions), and the inference time on CPU and GPU in ms, using batches of 32 images on reasonably powerful hardware.

As you can see, larger models are generally more accurate, but not always; for example, EfficientNetB2 outperforms InceptionV3 both in size and accuracy.

Class name	Size (MB)	Top-1 acc	Top-5 acc	Params	CPU (ms)	GPU (ms)
MobileNetV2	**14**	71.3%	90.1%	**3.5M**	25.9	3.8
MobileNet	16	70.4%	89.5%	4.3M	**22.6**	**3.4**
NASNetMobile	23	74.4%	91.9%	5.3M	27.0	6.7
EfficientNetB0	29	77.1%	93.3%	5.3M	46.0	4.9
EfficientNetB1	31	79.1%	94.4%	7.9M	60.2	5.6
EfficientNetB2	36	80.1%	94.9%	9.2M	80.8	6.5
EfficientNetB3	48	81.6%	95.7%	12.3M	140.0	8.8
EfficientNetB4	75	82.9%	96.4%	19.5M	308.3	15.1
InceptionV3	92	77.9%	93.7%	23.9M	42.2	6.9
ResNet50V2	98	76.0%	93.0%	25.6M	45.6	4.4
EfficientNetB5	118	83.6%	96.7%	30.6M	579.2	25.3
EfficientNetB6	166	84.0%	96.8%	43.3M	958.1	40.4
ResNet101V2	171	77.2%	93.8%	44.7M	72.7	5.4
InceptionResNetV2	215	80.3%	95.3%	55.9M	130.2	10.0
EfficientNetB7	256	**84.3%**	**97.0%**	66.7M	1578.9	61.6

Classification and Localization

Localizing an object in a picture can be expressed as a regression task: to predict a bounding box around the object, a common approach is to predict the horizontal and vertical coordinates of the object's center, as well as its height and width.

This means we have four numbers to predict. It does not require much change to the model; we just need to add a second dense output layer with four units, and it can be trained using the MSE loss:

Classification and Localization

But now we have a problem: the flowers dataset does not have bounding boxes around the flowers. So, we need to add them ourselves. This is often one of the hardest and most costly parts of a machine learning project: getting the labels. It’s a good idea to spend time looking for the right tools.

To annotate images with bounding boxes, you may want to use an open source image labeling tool like VGG Image Annotator, LabelImg, OpenLabeler, or ImgLab, or a commercial tool like LabelBox or Supervisely. You may also want to consider crowdsourcing platforms such as Amazon Mechanical Turk if you have a very large number of images to annotate.

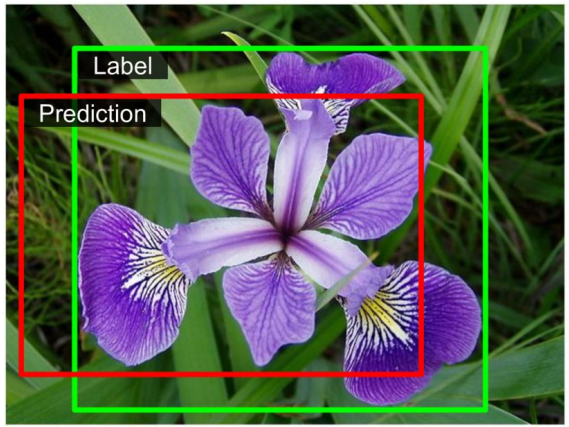
It may be preferable to do it yourself: with the right tools, it will only take a few days, and you’ll also gain a better understanding of your dataset and task.

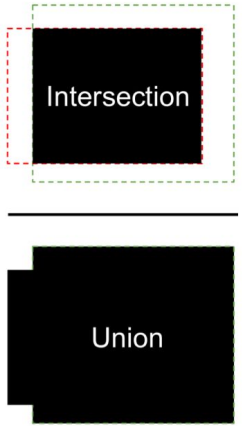
Classification and Localization

For now we will assume there is a single bounding box per image.

The MSE often works fairly well as a cost function to train the model, but it is not a great metric to evaluate how well the model can predict bounding boxes.

The most common metric for this is the *intersection over union* (IoU): the area of overlap between the predicted bounding box and the target bounding box, divided by the area of their union.
In Keras, it is implemented by the tf.keras.metrics.MeanIoU class.



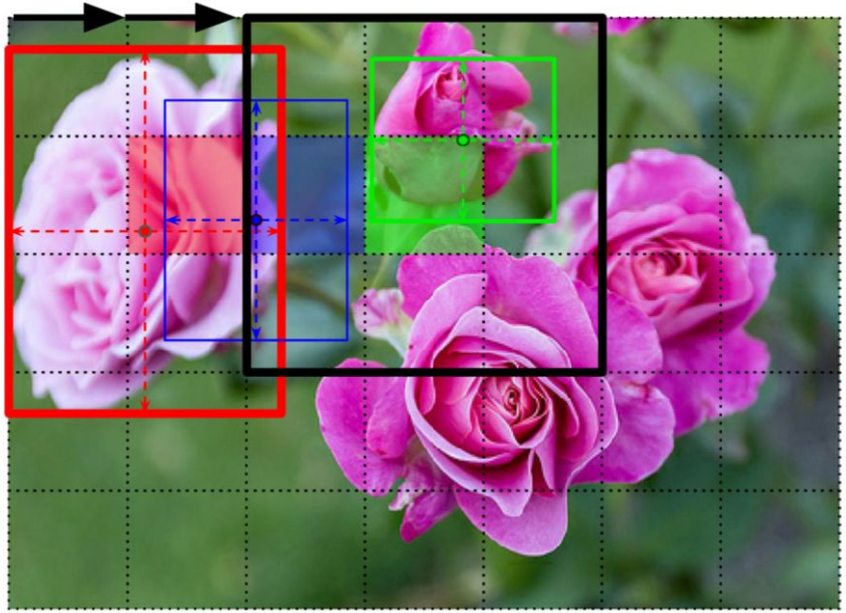


Object Detection

The task of classifying and localizing multiple objects in an image is called *object detection*.

Until a few years ago, a common approach was to take a CNN that was trained to classify and locate a single object roughly centered in the image, then slide this CNN across the image and make predictions at each step.

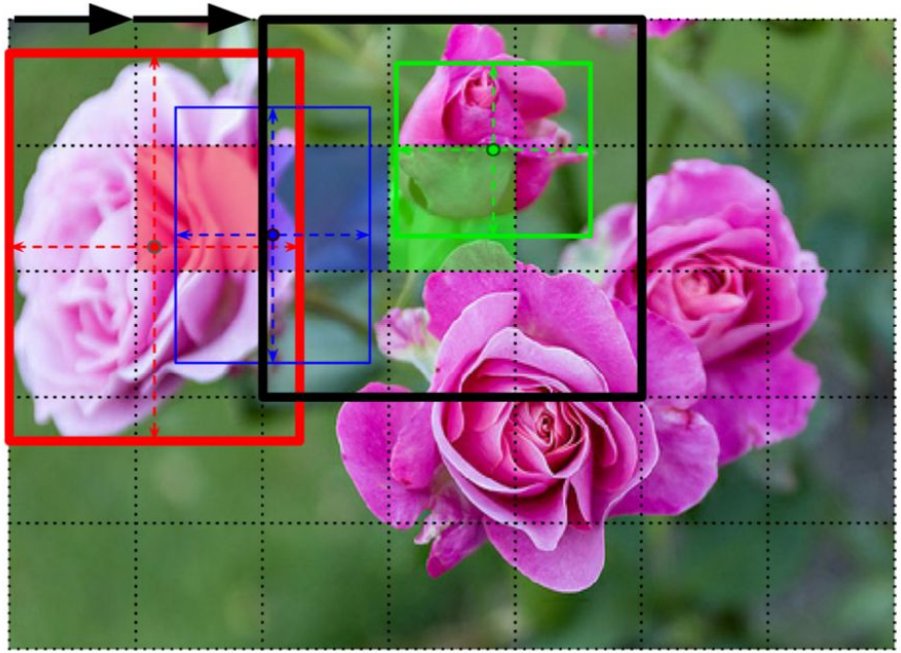
In this example, the image was chopped into a 5 × 7 grid, and we see a CNN - the thick black rectangle - sliding across all 3 × 3 regions and making predictions at each step.



Object Detection

This simple approach to object detection works pretty well, but it requires running the CNN many times (15 times in this example), so it is quite slow.

There is a much faster way to slide a CNN across an image: using a *fully convolutional network* (FCN).



Fully Convolutional Networks

The idea of FCNs was first introduced by Jonathan Long et al., for semantic segmentation (the task of classifying every pixel in an image according to the class of the object it belongs to).

The authors pointed out that you could replace the dense layers at the top of a CNN with convolutional layers.

Fully Convolutional Networks

The FCN approach is *much* more efficient, since the network only looks at the image once.

In fact, *You Only Look Once* (YOLO) is the name of a very popular object detection architecture, which we'll look at next.

You Only Look Once

YOLO is a fast and accurate object detection architecture proposed in 2015.

It is so fast that it can run in real time on a video.

For each grid cell, YOLO only considers objects whose bounding box center lies within that cell.

Object Tracking

Object tracking is a challenging task: objects move, they may grow or shrink as they get closer to or further away from the camera, their appearance may change as they turn around or move to different lighting conditions or backgrounds, they may be temporarily occluded by other objects, and so on.

One of the most popular object tracking systems is DeepSORT. It is based on a combination of classical algorithms and deep learning:

- It uses *Kalman filters* to estimate the most likely current position of an object given prior detections, and assuming that objects tend to move at a constant speed.
- It uses a deep learning model to measure the resemblance between new detections and existing tracked objects.

Semantic Segmentation

Each pixel is classified according to the class of the object it belongs to (e.g., road, car, pedestrian, building, etc.).

Note that different objects of the same class are *not* distinguished. For example, all the bicycles on the right side of the segmented image end up as one big lump of pixels.

Semantic Segmentation

The main difficulty in this task is that when images go through a regular CNN, they gradually lose their spatial resolution (due to the layers with strides greater than 1); so, a regular CNN may end up knowing that there's a person somewhere in the bottom left of the image, but it will not be much more precise than that.

Semantic Segmentation

Just like for object detection, there are many different approaches to tackle this problem, some quite complex.

However, a fairly simple solution was proposed based on fully convolutional networks. The authors start by taking a pretrained CNN and turning it into an FCN.

Instance Segmentation

Instance segmentation is similar to semantic segmentation, but instead of merging all objects of the same class into one big lump, each object is distinguished from the others (e.g., it identifies each individual bicycle).

The field is moving fast, though so if you want to try the latest and greatest models, please check out the state-of-the-art section of *https:// paperswithcode.com*.

Summary

As you can see, the field of deep computer vision is vast and fast-paced, with all sorts of architectures popping up every year. Almost all of them are based on convolutional neural networks, but since 2020 another neural net architecture has entered the computer vision space: transformers.

Summary

The progress made over the last decade has been astounding, and researchers are now focusing on harder and harder problems, such as

- *adversarial learning* (which attempts to make the network more resistant to images designed to fool it),
- *explainability* (understanding why the network makes a specific classification),
- realistic *image generation*,
- *single-shot learning* (a system that can recognize an object after it has seen it just once),
- predicting the next frames in a video,
- combining text and image tasks, and more.