Machine Vision

09P9M80/D82 - Aprendizagem Profunda

2024.2 - Prof. Pedro Moura

Machine Vision

- We have previously seen a high-level overview of particular applications of deep learning.
- With the foundational, low-level theory we've covered, we are now well positioned to work through specialized content across a range of application areas.
  - Primarily via hands-on example code.
- In this class, we will see Convolutional Neural Networks and apply them to Machine Vision tasks.

Convolutional Neural Networks

- A **Convolutional Neural Network** (also known as a ConvNet or a CNN) is an artificial neural network that features one or more **convolutional layers** (conv layers).
- This layer type enables a deep learning model to **efficiently process spatial patterns**.
- This property makes conv layers especially especially effective in computer vision applications.

The Two-Dimensional Structure of Visual Imagery

- In previous examples using MNIST digits, we converted image data into one-dimensional arrays of numbers.
  - To feed them into a dense layer.

- We transformed 28 x 28-pixel grayscale images to 784-element one-dimensional arrays: **flattening**.
  - We also divided by 255 in order to scale everything to [0, 1].

- This was a necessary step in the context of a **dense, fully connected network**.
  - We needed to flatten the 784 pixel values so that each one could be fed into a neuron of the first hidden layer.

- **Collapse of a 2D-image into 1D → substantial loss of meaningful visual image structure.**

The Two-Dimensional Structure of Visual Imagery

- If we printed an MNIST digit here as a 784-pixel long stream in shades of gray, we wouldn't be able to identify the digit.
- Instead, humans perceive visual information in a two-dimensional form (actually three-dimensional).
- Our ability to recognize what we're looking at is inherently tied to the **spatial relationships between the shapes and colors** we perceive.

Computational Complexity

- Besides the loss of structure when we collapse an image, a second consideration when piping images into a dense network is **computational complexity**.

- MNIST images are very small: 28 x 28 pixels with only one **channel**.
  - MNIST digits are monochromatic.
  - Images in full color → three channels (red, green and blue).

- Passing MNIST image information into a dense layer → **785 parameters per neuron** (784 weights for each of the pixels + neuron's bias).

Computational Complexity

- If we were handling a moderately sized image, however (e.g., 200 x 200-pixel, full-color RGB image) → number of parameters increases dramatically.

- In this case, we'd have three color channels, each with 40,000 pixels, corresponding to a total of 120,001 parameters **per neuron in a dense layer**.
  - 200 pixels x 200 pixels x 3 color channels + 1 bias = 120,001 parameters.

- Modest number of neurons in the dense layer (e.g., 64) → nearly 8 million parameters associated with the first hidden layer.
  - 64 neurons x 120,001 parameters per neuron = 7,680,064 parameters.

- This image is only 200 x 200 pixels - that's barely 0.4MP (most modern smartphones have 12MP or greater camera sensors).

Computational Complexity

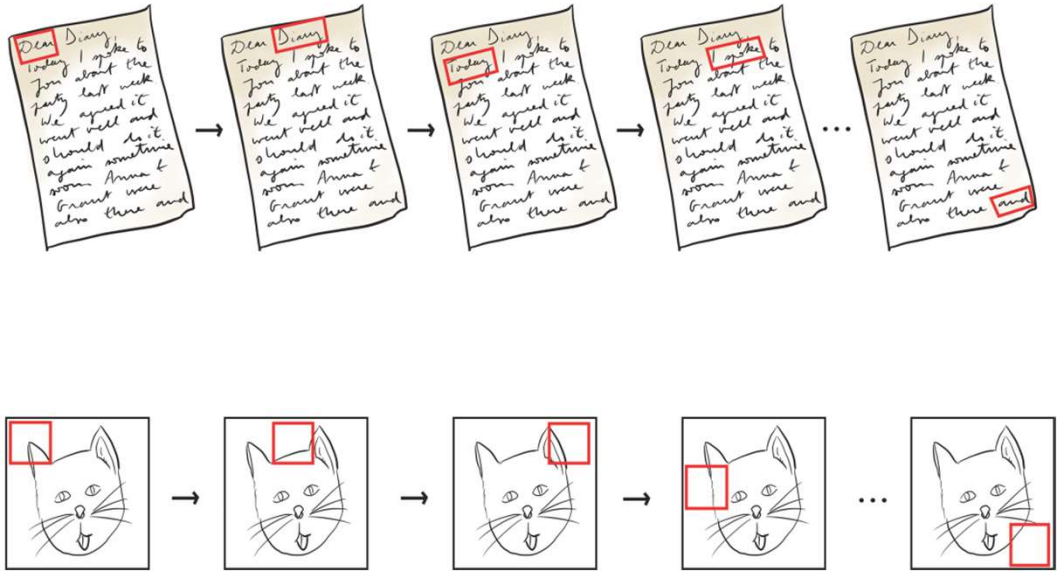
- One shall note that, generally, machine vision tasks don't need to run on high-resolution images in order to be successful.

- It should be clear: Images can contain a very large number of data points, and **using these in a naïve, fully connected manner will explode the neural network's compute power requirements.**

Convolutional Layers

- Convolutional layers consist of sets of **kernels**, which are also known as **filters**.
- Each of these kernels is a small window (called a **patch**) that scans across the image from top left to bottom right → **convolutional operation**.
- Let's see an illustration to better understand.

Convolutional Layers



When reading a page of a book written in English, we begin in the top-left corner and read to the right. Every time we reach the end of a row of text, we progress to the next row. In this way, we eventually reach the bottom-right corner, thereby reading all of the words on the page. Analogously, the kernel in a convolutional layer behaves in the same way.

Convolutional Layers

- Kernels are made up of weights, which are learned through backpropagation (as in dense layers).

- Kernels can range in size, but a typical size is 3 x 3, and we use that in the examples we will see in this class.
  - Another typical size is 5 x 5, with kernels larger than that used infrequently.

- For the monochromatic MNIST digits → this 3 x 3-pixel window would consist of 3 x 3 x 1 weights
  - 9 weights, for a total of 10 parameters.
  - Like an artificial neuron in a dense layer, every convolutional filter has a bias term **b**.

Convolutional Layers

- Full-color RGB images → kernel covering the same number of pixels would have three times as many weights – 3 x 3 x 3 of them.
  - A total of 27 weights and 28 parameters.

- The kernel occupies discrete positions across an image as it convolves.

- Sticking with the 3 x 3 kernel size for this explanation:
  - During forward propagation, a multidimensional variation of the "most important equation in this book" – **w . x + b** - is calculated at each position that the kernel occupies as it convolves over the image.

Convolutional Layers

- Let's refer to these 3 x 3 window of pixels and the 3 x 3 kernel:

.01	.09	.22
-1.36	.34	-1.59
.13	-.69	1.02

kernel weights

•

.53	.34	.06
.37	.82	.01
.62	.91	.34

pixel input

A 3×3 kernel and a 3×3-pixel window.

Convolutional Layers

- We can demonstrate the calculation of the weighted sum **w** . **x** in which products are calculated elementwise based on the alignment of vertical and horizontal locations.

- It's helpful to imagine the kernel superimposed over the pixel values.

$$\begin{aligned} \boldsymbol{w} \cdot \boldsymbol{x} &= .01 \times .53 + .09 \times .34 + .22 \times .06 \\ &+ -1.36 \times .37 + .34 \times .82 + -1.59 \times .01 \\ &+ .13 \times .62 + -.69 \times .91 + 1.02 \times .34 \\ &= -0.3917 \end{aligned}$$

Convolutional Layers

- We add some bias term **b** (say, 0.20) to arrive at **z**:

$$
\begin{aligned}
z &= w \cdot x + b \\
&= -0.39 + b \\
&= -0.39 + 0.20 \\
&= -0.19
\end{aligned}
$$

Convolutional Layers

- With **z**, we can at last calculate an activation value **a** by passing **z** through the activation function of our choice, say the tanh function or the ReLU function.

- The fundamental operation hasn't changed relative to the artificial neuron mathematics we've learned.

- Convolutional kernels have **weights**, **inputs** and **bias**.
  - A weighted sum of these is produced using our most important equation.
  - The resulting **z** is passed through some nonlinear function to produce an **activation**.

- What has changed is that there isn't a weight for **every input**, but rather a discrete kernel with 3 x 3 weights.

Convolutional Layers

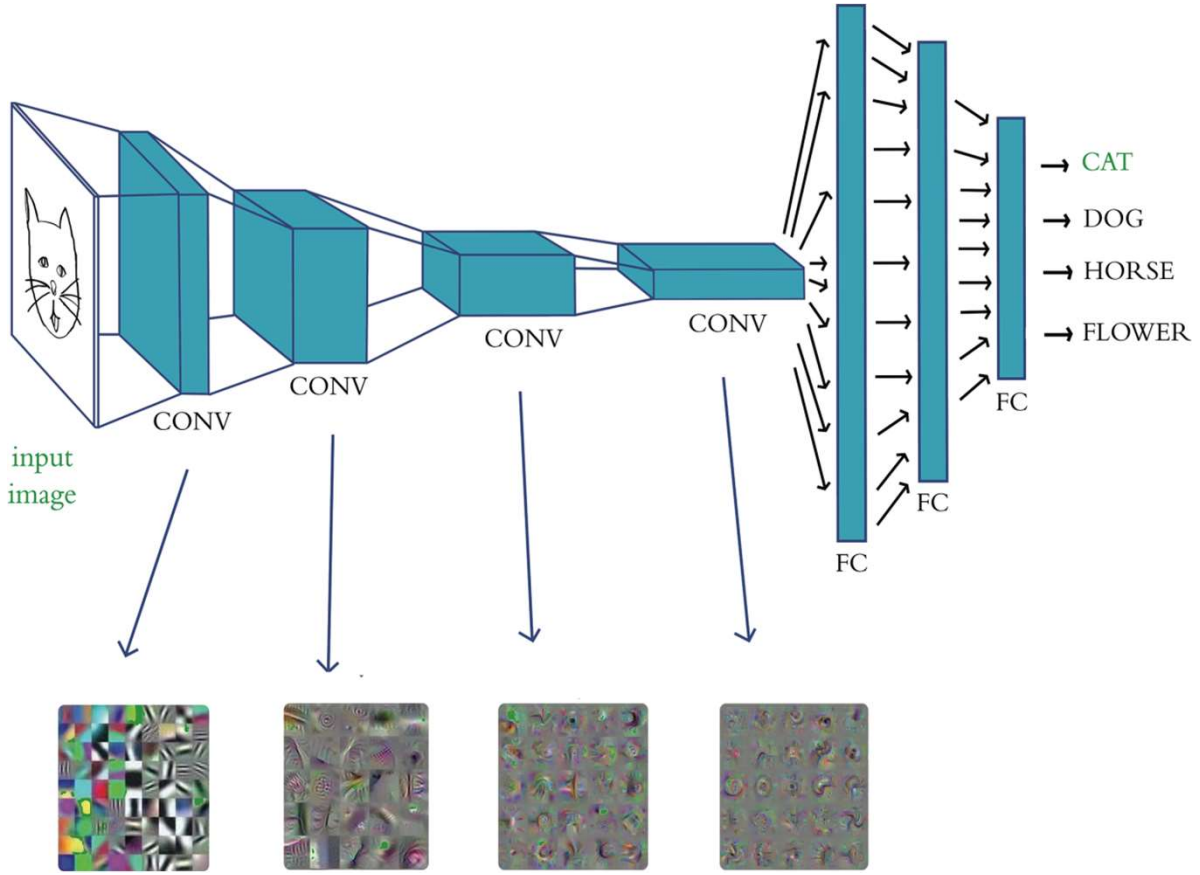
- These weights do not change as the kernel convolves; instead they're **shared** across **all of the inputs**.
- **A convolutional layer can have orders of magnitude fewer weights than a fully connected layer.**
- Another important point: like the inputs, the outputs from this kernel (all of the activations) are also arranged in a two-dimensional array.

Multiple Filters

- Typically, we have multiple filters in a given convolutional layer.
- Each filter enables the network to learn a representation of the data at a given layer in a unique way.
- Analogous to Hubel and Wiesel's simple cells → if the first hidden layer in our network is a convolutional layer, it might contain a kernel that responds **optimally to vertical lines**.
- Whenever it convolves (slides over) a vertical line in an input image, it produces a large activation (**a**) value.

Multiple Filters

- Additional kernels in this layer can learn to represent other simple spatial features such as horizontal lines and color transitions



Multiple Filters

- This is how these kernels came to be known as **filters**:
  - they scan over the image and filter out the location of specific features, producing high activations when they come across the pattern, shape, and/or color they are specially tuned to detect.

- They function as highlighters, producing a **two-dimensional array of activations** that indicate **where** that filter's particular feature exists in the original image.

- For this reason, the output from a kernel is referred to as an **activation map**.

Multiple Filters

- Subsequent convolutional layers receive these activation maps as their inputs.

- As the network gets deeper, the filters in the layers react to increasingly complex combinations of these simple features, learning to represent increasingly abstract spatial patterns.
  - **Eventually building a hierarchy from simple lines and colors up to complex textures and shapes.**

- Later layers within the network have the capacity to recognize whole objects or even, say, to distinguish an image of a Great Dane from that of a Yorkshire Terrier.

Multiple Filters

- The number of filters in the layer is also a hyperparameter that we configure.
- There is a Goldilocks sweet spot for filter number.
- Rules of thumb:
  - Having a larger number of kernels facilitates the identification of **more-complex features**, so consider the complexity of the data and the problem you're solving (**Beware of the computation efficiency**).
  - If a network has multiple convolutional layers, the optimal number of kernels for a given layer could vary quite a bit from layer to layer.

Multiple Filters

- Keep in mind that early layers identify simple features, whereas later layers identify complex recombinations of these simple features.

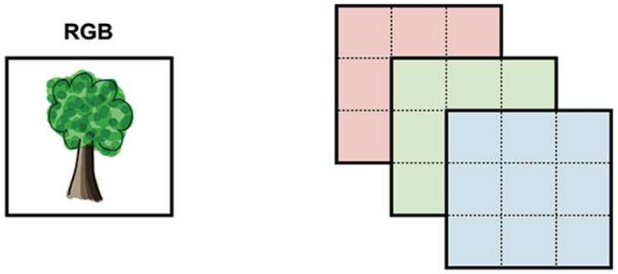
- Common approach for machine vision: is to have **many more kernels in later convolutional layers relative to early convolutional layers**.

- **Strive to minimize computational complexity:** Consider using the smallest number of kernels that facilitates a low cost on your validation data.
  - If doubling the number of kernels (say, from 32 to 64 to 128) in a given layer significantly decreases your model's validation cost, then consider using the higher value.
  - If halving the number of kernels (say, from 32 to 16 to 8) in a given layer doesn't increase your model's validation cost, then consider using the smaller value.

A Convolutional Example

- Convolutional layers are a nontrivial departure from the simpler fully connected layers.
- To better understand the way the pixel values and weights combine to produce feature maps, we will see a detailed contrived example.
- Imagine we're convolving over a single RGB image that's 3 x 3 pixels in size.
  - In Python, those data are stored in a [3,3,3] array.

A Convolutional Example



Images:	Red					Green					Blue				
	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.00	0.20	0.40	0.30	0.00	0.00	0.60	0.60	0.90	0.00	0.00	0.80	0.40	0.90	0.00
	0.00	0.30	0.90	0.60	0.00	0.00	0.40	0.70	0.40	0.00	0.00	0.30	0.10	0.60	0.00
	0.00	0.90	0.10	0.20	0.00	0.00	0.70	0.50	0.30	0.00	0.00	0.80	0.60	0.40	0.00
	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

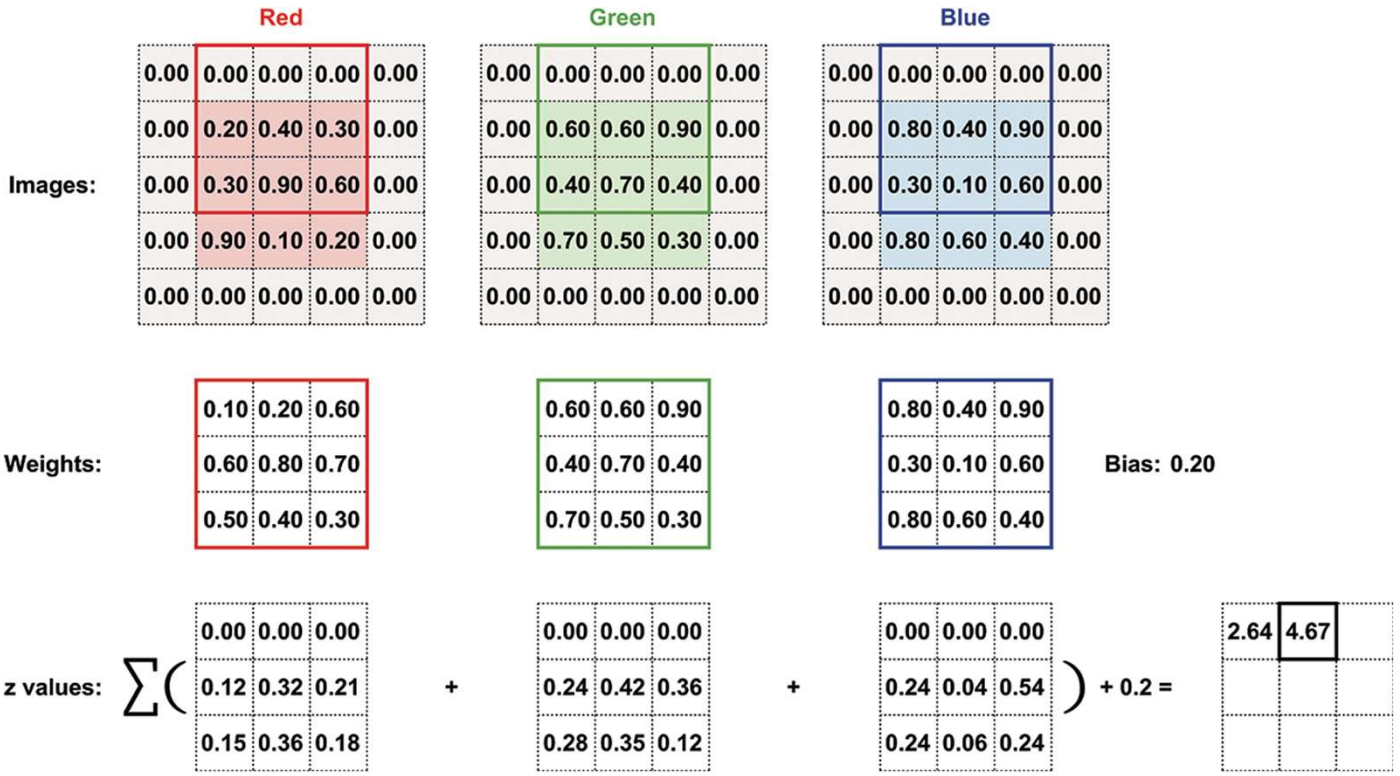
Weights:										Bias: 0.20
	0.10	0.20	0.60	0.60	0.60	0.90	0.80	0.40	0.90	
	0.60	0.80	0.70	0.40	0.70	0.40	0.30	0.10	0.60	
	0.50	0.40	0.30	0.70	0.50	0.30	0.80	0.60	0.40	

z value: $\sum($				+				+				) + 0.2 =			
	0.00	0.00	0.00		0.00	0.00	0.00		0.00	0.00	0.00		2.64		
	0.00	0.16	0.28		0.00	0.42	0.24		0.00	0.08	0.24				
	0.00	0.12	0.27		0.00	0.20	0.21		0.00	0.18	0.04				

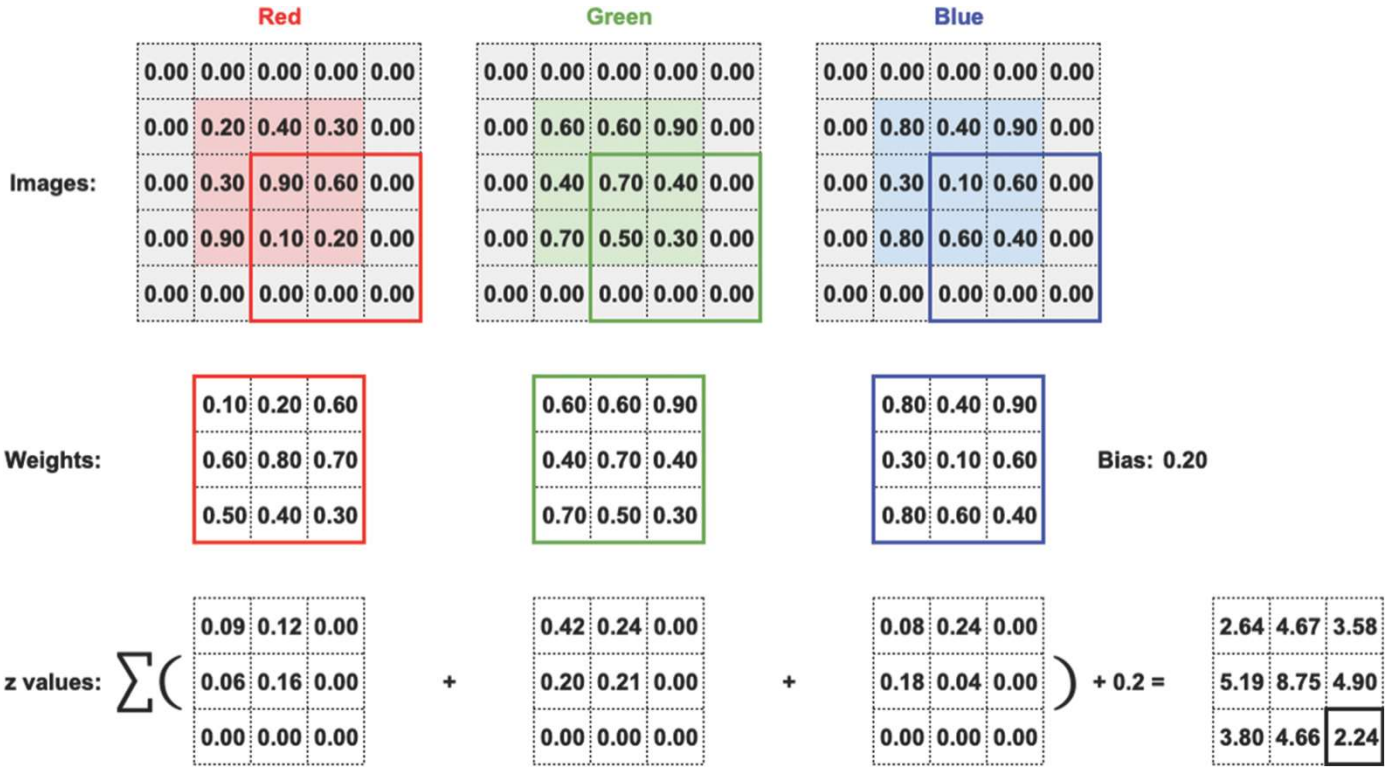
A Convolutional Example

- Shown in the middle of the figure are the 3 x 3 arrays for each of the three channels: red, green, and blue.
- The image has been padded with zeros on all four sides.
- We'll discuss more about padding shortly, but for now all you need to know is this: padding is used to ensure that the resulting feature map has the same dimensions as the input data.
- We chose a kernel size of 3 x 3, and given that there are three channels in the input image the weights matrix will be an array with dimensions [3,3,3] shown here individually.

A Convolutional Example



A Convolutional Example



A Convolutional Example

- This process is repeated for every possible filter position, and the **z**-value that was calculated for each of these nine positions is shown in the bottom-right corner.

- To convert this 3 x 3 map of **z**-values into a corresponding 3 x 3 activation map, we pass each **z**-value through an activation function, such as the ReLU function.

- A single convolutional layer nearly has multiple filters, each producing its own two-dimensional activation map, **activation maps have an additional depth dimension**.

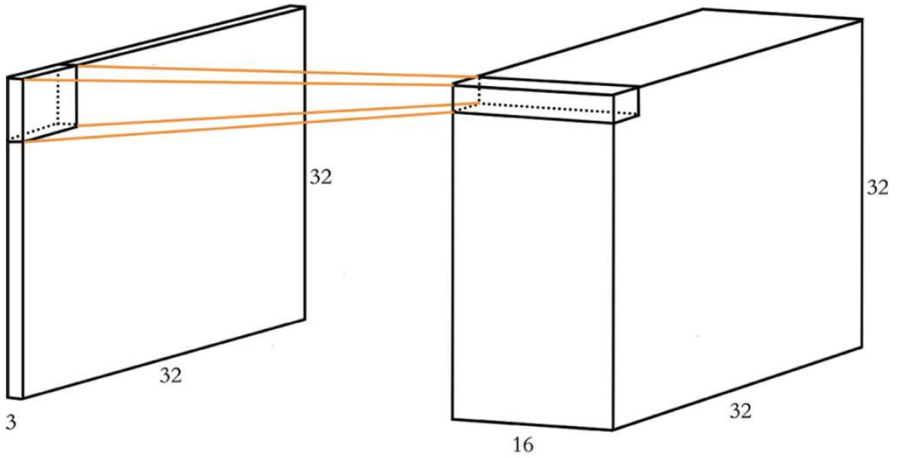
A Convolutional Example

- Each of these kernel “channels” in the activation map represents a feature that that particular kernel specializes in recognizing, such as an edge (a straight line) at a particular orientation

- Let’s see how the calculation of activation values **a** from the input image build up a three-dimensional activation map.

- The convolutional layer that produced the activation map has 16 kernels, → activation map with a depth of 16 “channels” (we’ll call **slices**).

A Convolutional Example



A graphical representation of the input array (left; represented here is a three-channel RGB image of size 32×32 with the kernel patch currently focused on the first—i.e., top-left—position) and the activation map (right). There are 16 kernels, resulting in an activation map with a depth of 16. Each position a given kernel occupies as it convolves over the input image corresponds to one ***a*** value in the resulting activation map.

A Convolutional Example

- The kernel filter is positioned over the top-left corner of the input image.
- This corresponds to 16 activation values in the top-left corner of the activation map: one activation **a** for each of the 16 kernels.
- By convolving over all of the pixel windows in the input image from left to right and from top to bottom, **all of the values in the activation map are filled in.**
- **Note that we have only one activation map output for each convolutional kernel, independently of the number of channels.**
- Let's see an interactive demonstration of convolutional-filter calculations.

A Convolutional Example

- If the first of the 16 filters is tuned to respond optimally to vertical lines → the first slice of the activation map will highlight all the physical regions of the input image that contain **vertical lines**.
- If the second filter is tuned to respond optimally to horizontal lines → the second slice will highlight regions of the image that contain **horizontal lines**.
- **In this way, all 16 filters in the activation map can together represent the spatial location of 16 different spatial features.**

A Convolutional Example

- Where do weights for a given convolutional kernel come from?
  - In these examples, all of the parameter values have been contrived.
- In real-world convolutional layers, kernel weights and biases are initialized with **random values**.
- **Then they are learned through backpropagation, such as weights and biases are learned in dense layers.**
- As we have seen:
  - Earliest convolutional layers tend to become tuned to simple features (like straight lines at particular orientations).
  - Deep convolutional layers tend to specialize in representing complex features (such as a face, a clock, or a dog).

A Convolutional Example

- Let's see the video by Jason Yosinski and his colleagues that vividly demonstrates the specializations of convolutional kernels by ConvNet layer depth.
  - Link

Summary

- Recognize features in a position invariant manner; a single kernel can identify its cognate feature anywhere in the input data.
- They remain faithful to the two-dimensional structure of images, allowing features to be identified within their spatial context.
- They significantly reduce the number of parameters required for modeling image data, yielding higher computational efficiency.
- Ultimately, they perform machine vision tasks (e.g., image classification) more accurately.

Convolutional Filter Hyperparameters

- In contrast to dense layers, convolutional layers are inherently **not** fully connected.
- **There isn't a weight mapping every single pixel to every single neuron in the first hidden layer.**
- **Convolutional layers → handful of hyperparameters that dictate the number of weights and biases associated.**
- Hyperparameters:
  - Kernel size
  - Stride length
  - Padding

Kernel Size

- **Kernel size** is also known as **filter size** or **receptive field**.
- In our examples, the **kernel size** has been 3 pixels wide and 3 pixels tall.
- **This is a common size found to be effective across a broad range of machine vision applications in contemporary ConvNet architectures.**
- Kernel size of 5 x 5 is also popular.
- 7 x 7 is about as expensive as they ever get.

Kernel Size

- Kernel too large with respect to the image → **too many competing features in the receptive field**.
  - It would be challenging for the convolutional layer to learn effectively.

- Kernel too small with respect to the image (e.g., 2 x 2) → **it wouldn't be able to tune to any structures**, and that isn't helpful either.

Stride Length

- Stride refers to the size of the step that the kernel takes as it moves over the image.
- In the example, we used a stride length of 1 pixel, which is a frequently used option.
  - Another common choice is a 2-pixel stride and, less often, a stride of 3.
- **Anything much larger is likely to be suboptimal → the kernel might skip regions of the image that are of value to the model.**
- **Increasing the stride will yield an increase in speed → there are fewer calculations that need to be carried out.**

Stride Length

- As ever in deep learning, it's about finding a balance (sweet spot) between these effects.
- We recommend a stride of 1 or 2, while avoiding anything larger than 3.

Padding

- Padding plays handily with stride to keep the calculations of a convolutional layer in order.
- Let's suppose you had a 28 x 28 MNIST digit and a 5 x 5 kernel.
- With a stride of 1, there are 24 x 24 "positions" for the kernel to move through before it bumps up against the edges of the image.
  - The activation map output by the layer is slightly smaller than the input.
- **Solution: you can simply pad the image with zeros around the edges.**
  - So as to produce an activation map that is the exact same size as the input image.
- We had zero-padded images in the last example.

Padding

- In the case of the 28 x 28 image and the 5 x 5 kernel, padding with two zeros on each edge will produce a 28 x 28 activation map.

- Equation:

$$\text{Activation map} = \frac{D - F + 2P}{S} + 1$$

Padding

- Where:
  - **D** is the size of the image (either width or height, depending on whether you're calculating the width or height of the activation map).
  - **F** is the size of the filter.
  - **P** is the amount of padding.
  - **S** is the stride length.

Padding

- Using a padding of 2, we can calculate that the output volume is 28 x 28:

$$\text{Activation map} = \frac{D - F + 2P}{S} + 1$$

$$\text{Activation map} = \frac{28 - 5 + 2 \times 2}{1} + 1$$

$$\text{Activation map} = 28$$

Padding

- Kernel size, stride and padding are interconnected.
  - We have to make sure these hyperparameters align when designing CNN architectures.
- The **hyperparameters must combine** to produce a valid activation map size (an integer value).
- Take, for example, a kernel size of 5 x 5 with a stride of 2 and no padding.
- This would result in a 12.5 x 12.5 activation map.

Padding

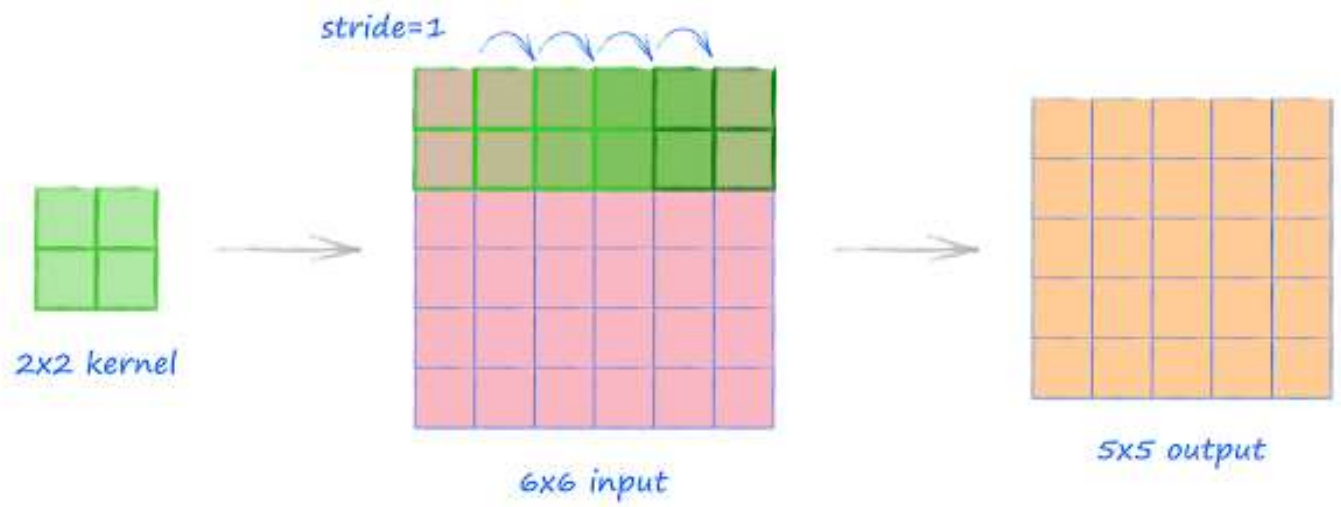
$$\text{Activation map} = \frac{D - F + 2P}{S} + 1$$

$$\text{Activation map} = \frac{28 - 5 + 0 \times 2}{2} + 1$$

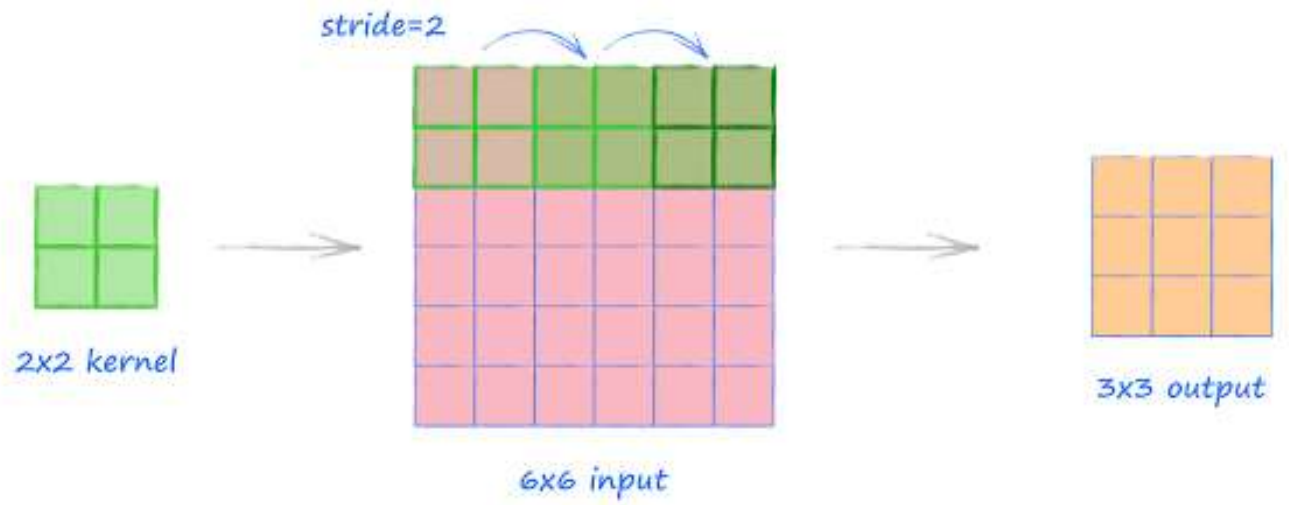
$$\text{Activation map} = 12.5$$

- **There is no such thing as a partial activation value → a convolutional layer with these dimensions would simply not be computable.**

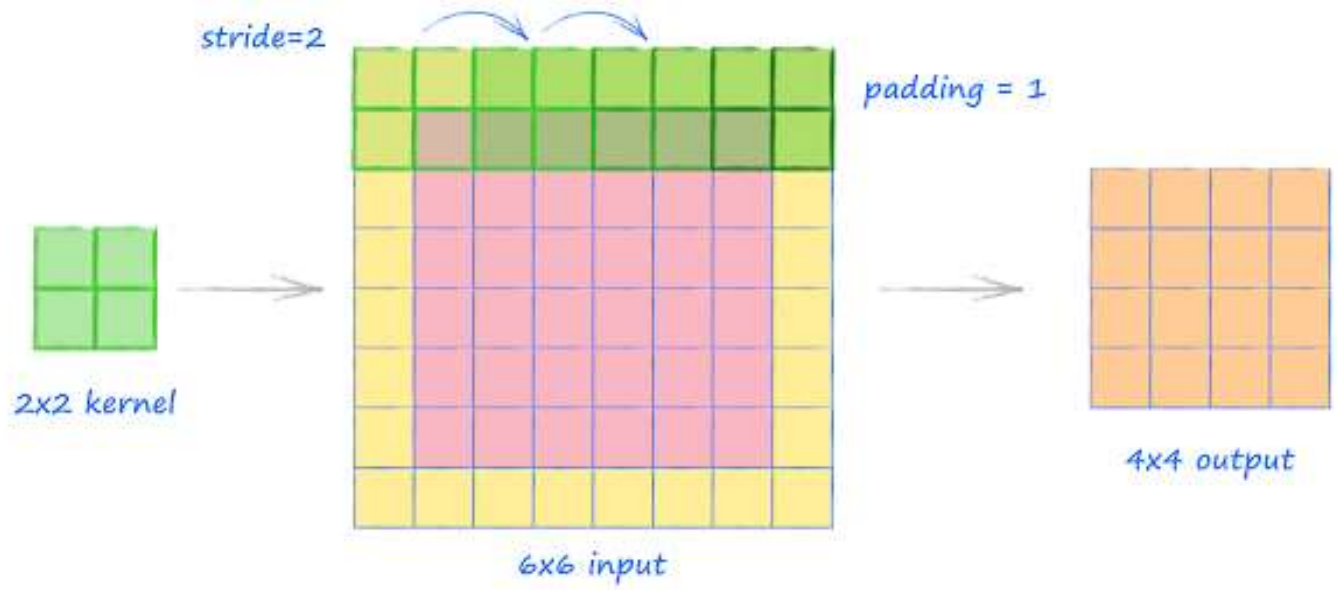
Padding



Padding



Padding



Pooling Layers

- Convolutional layers frequently work in tandem with another layer type: **pooling layers**.
- This layer type serves to **reduce** the **overall count of parameters** in a network as well as to **reduce complexity**.
- **Speeding up computation and helping to avoid overfitting.**

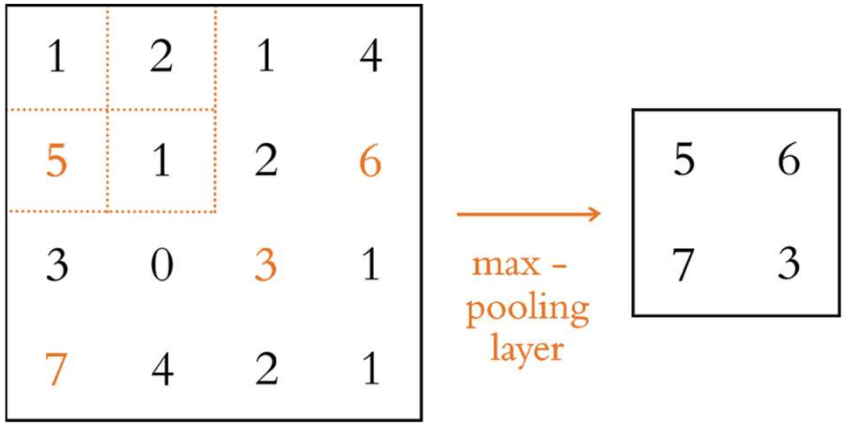
Pooling Layers

- A convolutional layer can have any number of kernels.
- Each of these kernels produces an activation map.
- The output from a convolutional layer is a **threedimensional array of activation maps.**
  - With the depth dimension of the output corresponding to the number of filters in that convolutional layer.
- **The pooling layer reduces these activation maps spatially, while leaving the depth of the activation maps intact.**

Pooling Layers

- Any given pooling layer has a filter size and a stride length.
- Also like a convolutional layer, the pooling layer slides over its input.
- At each position it occupies, the pooling layer applies a **data-reducing operation**.
- These layers most often use the **`max`** operation → these are termed **max-pooling layers**.
- They retain the largest value (the **maximum activation**) within the receptive field while discarding the other values.

Pooling Layers



With a 2×2-sized filter, the layer retains only the largest of four input values (e.g., the orange “5” in the 2 x 2 hatch-marked top-left corner). With a 2×2 stride, the resulting output from this max-pooling layer has one-quarter of the volume of its input: a 2 x 2 activation map.

Pooling Layers

- Typically, a pooling layer has a filter size of 2 x 2 and a stride length of 2.
  - This the default recommendation.
  - However, they are hyperparameters that you can experiment with, if desired.
- At each position:
  - The pooling layer evaluates four activations.
  - It retains only the maximum value.
  - Thereby downsampling the activations by a factor of 4.
- This pooling operation happens independently for each depth slice in the threedimensional array.
  - A 28 x 28 activation map with a depth of 16 slices would be reduced to a 14 x 14 activation map **but it would retain its full complement of 16 slices**.

Pooling Layers

- Other pooling variants: **average pooling** and **L2-norm pooling**.
  - Much less common relative to maxpooling.
- **Maxpooling** suits machine vision applications sufficiently accurately while requiring minimal computational resources.
- Alternative approach to pooling for reducing computational complexity is to **use a convolutional layer with a larger stride**.
- This can be handy for some specialized machine vision tasks (e.g., the **Generative Adversarial Networks**) that tend to perform better without pooling layers.

Pooling Layers

- Pooling layer during backpropagation:
  - The network keeps track of the index of the max value in each forward pass.
  - The gradient for that particular weight is backpropagated correctly.
  - This gradient is used to update the correct parameters.

LeNet-5 in Keras

- As we introduced the hierarchical nature of deep learning, we discussed the machine vision architecture called **LeNet-5**.
- We now use Keras to construct an MNIST digit-classifying model that is inspired by this landmark architecture.
- Some modern twists:
  - Computation is much cheaper today → we use more kernels in our convolutional layers. More specifically, we include 32 and 64 filters in the first and second convolutional layers, respectively, whereas the original LeNet-5 had only 6 and 16 in each.
  - Also thanks to cheap compute, we are subsampling activations only once (with a max-pooling layer), whereas LeNet-5 did twice.
  - We leverage innovations like ReLU activations and dropout, which had not yet been invented at the time of LeNet-5.

LeNet-5 in Keras

- Let's check it in the notebook called *Lenet in Keras*.
- We now use Keras to construct an MNIST digit-classifying model that is inspired by this landmark architecture.
- The first hidden layer in our LeNet-5-inspired network will be convolutional, so we can leave the images in the 28 x 28-pixel format.
  - Instead of reshaping the image to a one-dimensional array.

LeNet-5 in Keras

- Here we use convolutional layers (Conv2D) as our first two hidden layers.
- The integers 32 and 64 correspond to the number of filters we're specifying for the first and second convolutional layer, respectively.
- `kernel_size` is set to 3 x 3 pixels.
- We're using `relu` as our activation function.
- We're using the default stride length, which is 1 pixel (along both the vertical and the horizontal axes).

LeNet-5 in Keras

- We're using the default padding, which is `'valid'`.
- Our activation map will be 2 pixels shorter and 2 pixels narrower than the input to the layer (e.g., a 28 x 28-pixel input image shrinks to a 26 x 26 activation map).
- The alternative would be to specify the argument `padding='same'`, which would pad the input with zeros so that the output retains the same size as the input (a 28 x 28-pixel input image results in a 28 x 28 activation map).

LeNet-5 in Keras

- To our second hidden layer of neurons:
- `MaxPooling2D()` is used to reduce computational complexity, with `pool_size` set to 2 x 2 and the strides argument left at its default (None, which sets stride length equal to pool size).
  - We are reducing the volume of our activation map by three-quarters.
- `Dropout()` reduces the risk of overfitting to our training data.
- `Flatten()` converts the three-dimensional activation map output by `Conv2D()` to a one-dimensional array.
  - This enables us to feed the activations as inputs into a `Dense` layer, which can only accept one-dimensional arrays.

LeNet-5 in Keras

- The first convolutional layer learns to represent simple features like straight lines at a particular orientation.
- The second convolutional layer recombines those simple features into more-abstract representations.
- The intuition behind having a `Dense` layer as the third hidden layer in the network is that it allows the **spatial features** identified by the second convolutional layer to be **recombined in any way** that's optimal for **distinguishing classes of images**.
- The two convolutional layers learn to identify and label spatial features.
- **These spatial features are then fed into a dense layer that maps these spatial features to a particular class of images.**

LeNet-5 in Keras

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
conv2d_2 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_1 (MaxPooling2	(None, 12, 12, 64)	0
dropout_1 (Dropout)	(None, 12, 12, 64)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_1 (Dense)	(None, 128)	1179776
dropout_2 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 10)	1290

Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0

A summary of our LeNet-5-inspired ConvNet architecture.

LeNet-5 in Keras

- Cumulatively, the entire ConvNet has 1,199,882 parameters, **the vast majority (98.3 percent) of which are associated with the dense hidden layer**.
- Previously, our best result was attained by *Deep Net in Keras* - an accuracy of 97.87 percent on the validation set of MNIST digits.
- But here, the ConvNet inspired by LeNet-5 achieved 99.27 percent validation accuracy.
- This is fairly remarkable because the CNN wiped away 65.7 percent of the remaining error.
- Presumably these now correctly classified instances are some of the trickiest digits to classify.

LeNet-5 in Keras

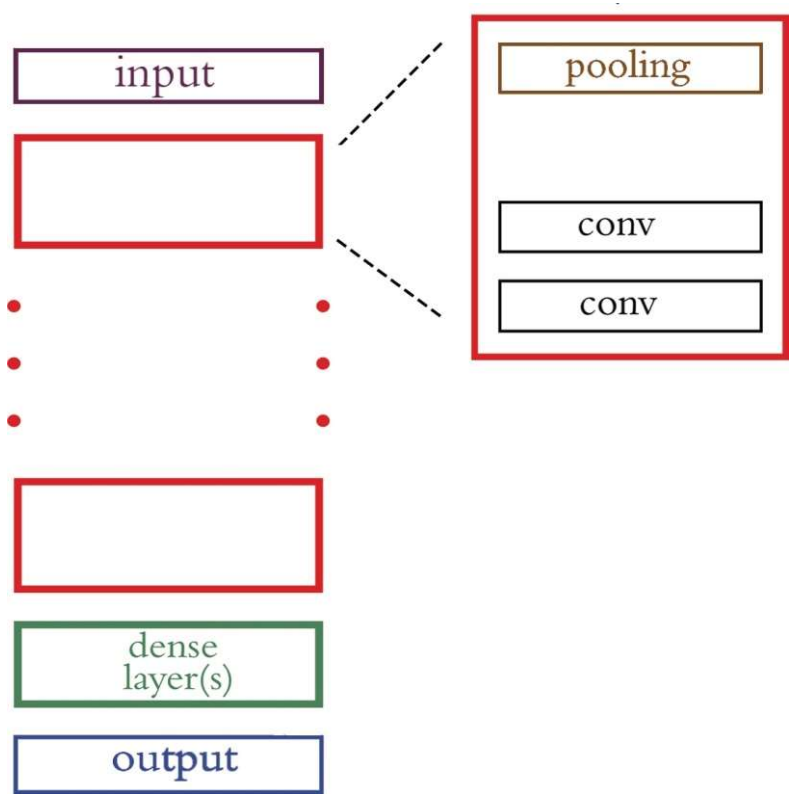
```
Epoch 9/10
60000/60000 [==============================] - 39s 654us/step - loss: 0.0276 - acc: 0.9911 - val_loss: 0.0260 - val_acc: 0.9927
```

Our LeNet-5-inspired ConvNet architecture peaked at a 99.27 percent validation accuracy following nine epochs of training, thereby outperforming the accuracy of the dense nets we trained earlier in the book.

AlexNet and VGGNet in Keras

- Routine in convolutional networks: **pair of convolutional layers followed by a max-pooling layer.**
- It is common to group convolutional layers (one to three) together with a pooling layer.
  - **These conv-pool layers can then be repeated several times.**
- Such CNN architectures regularly culminate in a **dense hidden layer (up to several ones)** and then the **output layer**.

AlexNet and VGGNet in Keras



A general approach to CNN design: A block (shown in red) of convolutional layers (often one to three of them) and a pooling layer is repeated several times. This is followed by one (up to a few) dense layers.

AlexNet and VGGNet in Keras

- The AlexNet is another architecture that features the convolutional layer block approach.
- In our *AlexNet in Keras* notebook, we emulate this structure.

AlexNet and VGGNet in Keras

- For this notebook, we moved beyond the MNIST digits to a dataset of larger-sized (224x224-pixel) images that are full-color (hence the 3 channels of depth).

- AlexNet used larger filter sizes in the earliest convolutional layers relative to what is popular today - for example, `kernel_size=(11, 11)`.

- The use of dropout in only the dense layers near the model output (and not in the earlier convolutional layers) is common.
  - Early convolutional layers enable the model to represent spatial features of images that generalize well beyond the training data.
  - However, a very specific recombination of these features, as facilitated by the dense layers, may be unique to the training dataset and thus may not generalize well to validation data.

AlexNet and VGGNet in Keras

- Following AlexNet being crowned the 2012 winner of the ILSVRC competition, deep learning models suddenly began to be used widely in the contest.
- Among these models, there has been a general trend toward making the neural networks deeper and deeper.
- For example, in 2014 the runnerup in the ILSVRC was VGGNet which follows the same repeated conv-pool-block structure as AlexNet.
  - VGGNet simply has more of them, and with smaller (all 3x3-pixel) kernel sizes.

Residual Networks

- As our example ConvNets (LeNet-5, AlexNet, VGGNet) suggest, there's a trend over time **toward deeper networks**.
- We are going to recapitulate the topic of vanishing gradients.
  - **The (often dramatic) slowing in learning that can occur as network architectures are deepened.**
- We then describe an imaginative solution that has emerged in recent years: **residual networks**.

Vanishing Gradients: The Bête Noire of Deep CNNs

- We alread learned that with more layers:
  - Models are able to learn a larger variety of relatively **low-level features** in the **early layers**.
  - **Increasingly complex abstractions** are made possible in the **later layers** via **nonlinear recombination**.

- **This approach has limits:**
  - If we continue to make our networks deeper (e.g., by adding more and more of the conv-pool blocks), they will eventually be **debilitated by the vanishing gradient problem**.

Vanishing Gradients: The Bête Noire of Deep CNNs

- Parameters in early layers of the network are far away from the cost function: the source of the gradient that is propagated backward through the network.

- As the error is backpropagated:
  - A larger and larger number of parameters contribute to the error.
  - **Thus each layer closer to the input gets a smaller and smaller update.**

- **The net effect is that early layers in increasingly deep networks become more difficult to train.**

Vanishing Gradients: The Bête Noire of Deep CNNs

- Consequence: it is commonly observed that as one increases the depth of a network, **accuracy increases up to a saturation point** and then later begins to **degrade as networks become excessively deep**.
- Imagine a shallow network that is performing well and let's copy those layers with their weights, and stack on new layers atop to make the model deeper.
- We might think that this new, deeper model would take the existing gains from the early pretrained layers and improve.
- If the new layers performed simple **identity mapping** (wherein they faithfully reproduced the exact results of the earlier layers), then we'd see no increase in training error.

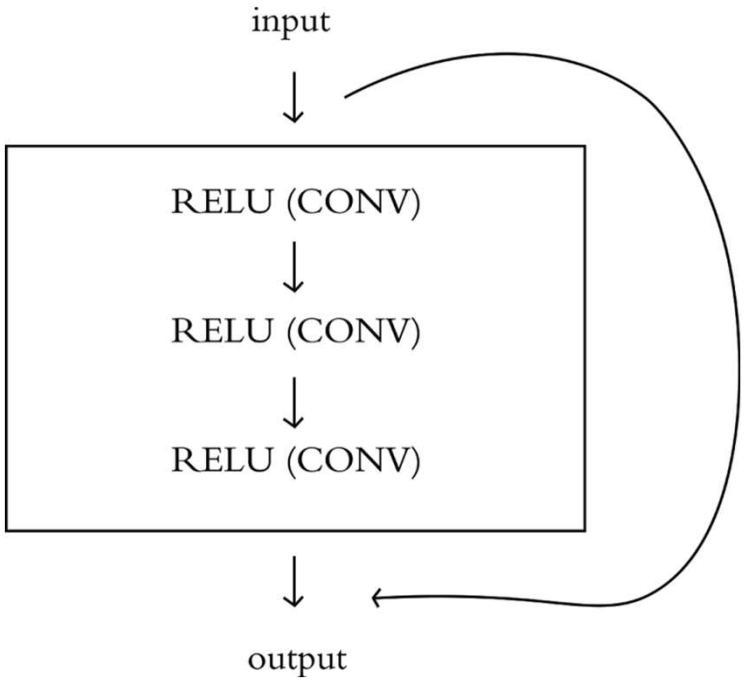
Vanishing Gradients: The Bête Noire of Deep CNNs

- **However, plain deep networks struggle to learn identity functions.**
- Thus, these new layers either **add new information and decrease the error**, or **they do not add new information (but also fail at identity mapping) and the error increases**.
- Given that adding useful information is an exceedingly rare outcome (relative to the baseline), beyond a certain point these extra layers will, **probabilistically, contribute to an overall degradation in performance**.

Residual Connections

- Residual networks (or ResNets, for short) rely on the idea of residual connections, which exist within so-called residual modules.
- **A residual module is a collective term for a sequence of convolutions, batch-normalization operations, and ReLU activations that culminates with a residual connection.**
- For the sake of simplicity here, we consider these various layers within a residual module to be a single, discrete unit.
- **A residual connection exists when the input to one such residual module is summed with its output to produce the final activation for that residual module.**

Residual Connections



A schematic representation of a residual module. Batch normalization and dropout layers are not shown, but may be included.

Residual Connections

- In other words, a residual module will receive some input a_{i-1} which is transformed by the convolutions and activation functions within the residual module to generate its output a_i.
- Subsequently, this output and the original input to the residual module are summed: $y_i = a_i + a_{i-1}$.
- An interesting feature emerges: if the residual module has an activation $a_i = 0$ - that is, it has learned nothing - the final output of the residual module will simply be the original input, since the two are summed.

Residual Connections

$$\begin{aligned} \boldsymbol{y}_i &= \boldsymbol{a}_i + \boldsymbol{a}_{i-1} \\ &= 0 + \boldsymbol{a}_{i-1} \\ &= \boldsymbol{a}_{i-1} \end{aligned}$$

- In this case, the residual module is effectively an **identity function**.
- **These residual modules either learn something useful and contribute to reducing the error of the network, or they perform identity mapping and do nothing at all.**

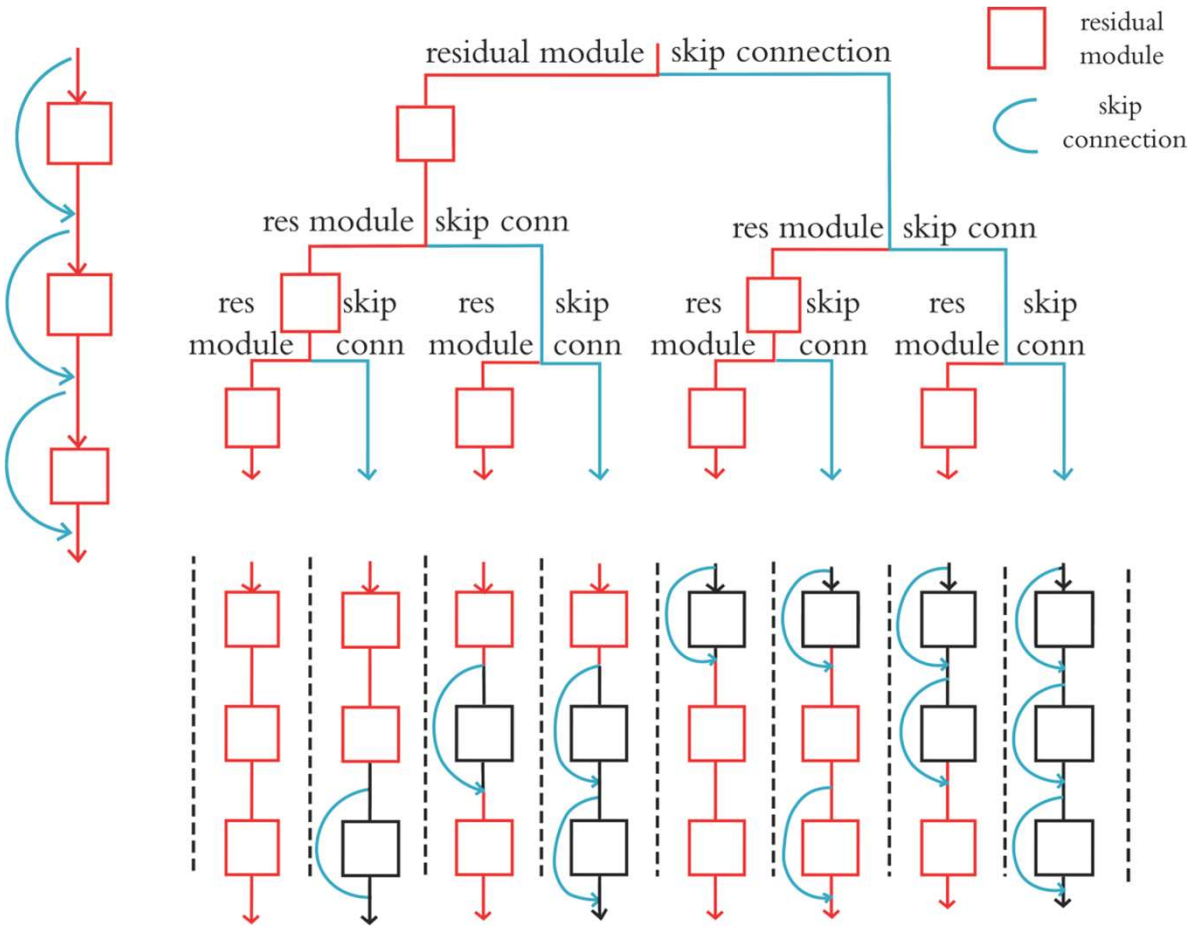
Residual Connections

- Because of this identity-mapping behavior, residual connections are also called **“skip connections”,** because they enable information to skip the functions located within the residual module.

- **Inherent multiplicity**: when several residual modules are stacked, later residual modules receive inputs that are increasingly complex combinations of the residual modules and skip connections from earlier in the network.

- A decision tree representation shows how, at each of the three residual modules in the network, information may either pass through the residual block or bypass it via a skip connection.

Residual Connections



Shown at left is the conventional representation of residual blocks within a residual network. Shown at right is an unraveled view, which demonstrates how, depending on which skip connections are used, the final path of information from input to output can be varied by the network.

Residual Connections

- **With only three residual modules there are eight possible paths the information can take.**

- In practice, the process is not commonly as binary as it is depicted in this figure.
  - The value of a_i is seldom 0, and therefore the output is usually some mix of the identity function and the residual module.

- Given this insight, residual networks can be thought of as complex combinations or ensembles of many shallower networks that are pooled at various depths.

ResNet

- The first deep residual network, ResNet, was introduced by Microsoft Research in 2015 and won first place in that year's ILSVRC image-classification competition.
- This makes ResNet the leader of the pack of deep learning algorithms that surpassed human performance at image recognition in 2015.
- ILSVRC has several machine vision competition categories, such as **object detection** and **image segmentation**.
- In 2015, ResNet took first place not only in the ILSVRC image-classification competition but in the object detection and image segmentation categories, too.

ResNet

- Still in 2015, ResNet was also recognized as champion of the detection and segmentation competitions involving an alternative image dataset called COCO, which is an alternative to the ILSVRC set.

- Given the broad sweep of machine vision trophies upon the invention of residual networks → **it's clear they were a transformative innovation**.

- **They managed to improve the existing networks by enabling much deeper architectures without the decrease in performance associated with those extra layers if they fail to learn useful information about the problem.**

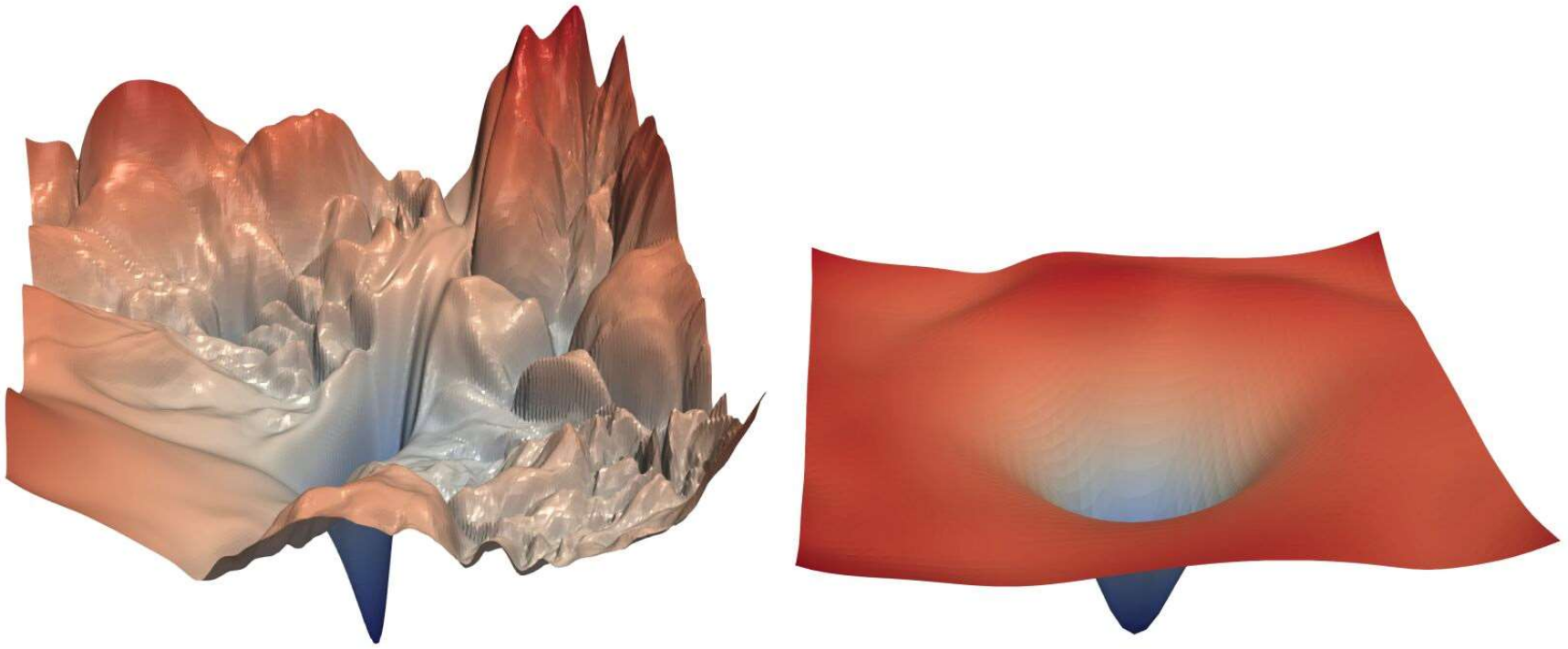
ResNet

- On the book adopted, the authors choose model architectures and datasets that are small enough to carry out training on even a modest laptop computer.

- Residual network architectures, as well as the datasets that make them worthwhile, do not fall into this category

- We are going to use **transfer learning** that provide us with resources to nevertheless take advantage of very deep architectures like ResNet with the model's parameters pretrained on massive datasets.

Residual Connections – Loss Surface



(a) without skip connections

(b) with skip connections

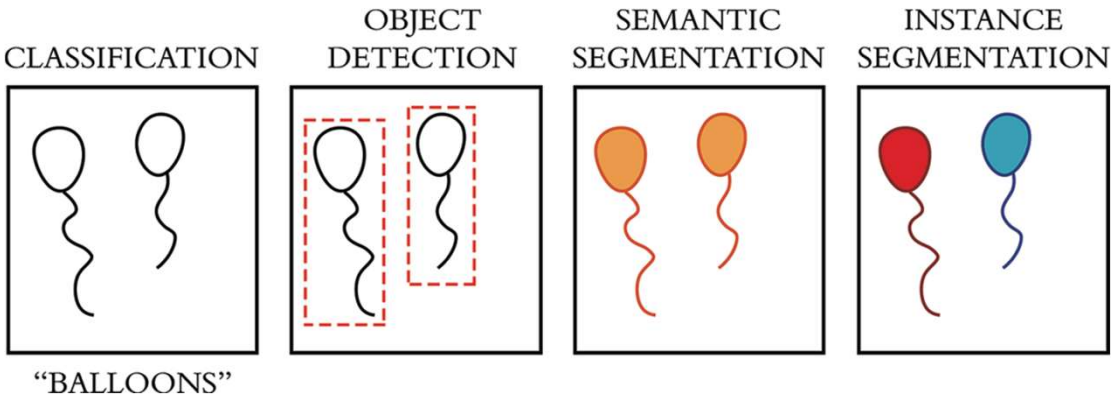
Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

Applications of Machine Vision

- We have learned about layer types that enable machine vision models to perform well.
- We've also discussed some of the approaches that are used to improve these models.
  - We've delved into some of the canonical machine vision algorithms of the past few years.
- Up to now, we've dealt with the problem of **image classification: identifying the main subject in an image**.
- We now turn our focus to other interesting applications of machine vision.

Applications of Machine Vision



These are examples of various machine vision applications. We have encountered classification previously in this chapter, but now we cover object detection, semantic segmentation, and instance segmentation.

Applications of Machine Vision

- The first is **object detection**, wherein the algorithm is tasked with drawing bounding boxes around objects in an image.

- Next is image segmentation.
  - **Semantic segmentation** identifies all objects of a particular class down to the pixel level.
  - **Instance segmentation** discriminates between different instances of a particular class, also at the pixel level.

Object Detection

- Imagine a photo of a group of people sitting down to dinner. There are several people in the image.
- There is a roast chicken in the middle of the table, and maybe a bottle of
- wine.
- If we desired an automated system that could predict what was served for dinner or to identify the people sitting at the table: **object detection**.
  - An image-classification algorithm would not provide that level of granularity
- Object detection has broad applications, such as detecting pedestrians in the field of view for autonomous driving, or for identifying anomalies in medical images.

Object Detection

- Object detection is divided into two tasks:
  - Detection (identifying **where** the objects in the image are).
  - Subsequently, classification (identifying **what** the objects are that have been detected).

- The pipeline has three stages:

  1) A region of interest must be identified.
  2) Automatic feature extraction is performed on this region.
  3) The region is classified.

- Seminal models—ones that have defined progress in this area—include R-CNN, Fast R-CNN, Faster R-CNN, and YOLO.

R-CNN

- R-CNN was proposed in 2013 by Ross Girshick and his colleagues at UC Berkeley.

- **The algorithm was modeled on the attention mechanism of the human brain, wherein an entire scene is scanned and focus is placed on specific regions of interest.**

- Girshick and his coworkers developed R-CNN to:

  1) Perform a selective search for **regions of interest** (ROIs) within the image.
  2) Extract features from these ROIs by using a CNN.
  3) Combine two “traditional” machine learning approaches - linear regression and support vector machines - to, respectively, refine the locations of bounding boxes and classify objects within each of those boxes.

R-CNN

- R-CNNs redefined the state of the art in object detection, achieving a massive gain in performance over the previous best model in the Pattern Analysis, Statistical Modeling and Computational Learning (PASCAL) Visual Object Classes (VOC) competition.
  - It is considered one of the gold standards for object-detection problems.
- This ushered in the era of deep learning in object detection.
- However, this model had some limitations:
  - **It was inflexible:** The input size was fixed to a single specific image shape.
  - **It was slow and computionally expensive:** both training and inference are multistage processes involving CNNs, linear regression models, and support vector machines.

Fast R-CNN

- To address the primary drawback of R-CNN - **its speed** - Girshick went on to develop Fast R-CNN.

- Chief innovation → during step 2 of the R-CNN, the CNN was unnecessarily being run multiple times, once for each region of interest.

- With Fast R-CNN:
  - The ROI search (step 1) is run as before.
  - In step 2, the CNN is given a single global look at the image, and the extracted features are used for all ROIs simultaneously.
  - A vector of features is extracted from the final layer of the CNN, which (for step 3) is then fed into a dense network along with the ROI.

Fast R-CNN

- This dense net learns to focus on only the features that apply to each individual ROI, culminating in two outputs per ROI:
- 1. A softmax probability output over the classification categories (what class the detected object belongs to).
- 2. A bounding box regressor (for refinement of the ROI's location).

- Fast R-CNN model has to perform feature extraction using a CNN only once for a given image: **reduced computational complexity**.

- As the name suggests, the reduced computational complexity of Fast R-CNN corresponds to speedier compute times.

- It also represents a single, unified model without the multiple independent parts of its predecessor.
  - **Bottleneck: the initial (ROI search) step of Fast R-CNN**.

Faster R-CNN

- These model architectures are clever works of innovation - **their names, however, are not.**
- Our third object-detection algorithm of note is Faster R-CNN, which is even swifter than Fast R-CNN.
- Faster R-CNN was revealed in 2015 by Shaoqing Ren and his coworkers at Microsoft Research
- To overcome the ROI-search bottleneck → leverage the feature activation maps from the model's CNN for this step, too.
- **Those activation maps contain a great deal of contextual information about an image.**

Faster R-CNN

These are examples of object detection (performed on four separate images by the Faster R-CNN algorithm). Within each region of interest—defined by the bounding boxes within the images—the algorithm predicts what the object within the region is.

Faster R-CNN

- Each map has two dimensions representing location → they can be thought of as literal **maps** of the locations of features within a given image.
- Convolutional layer with 16 filters + activation map it outputs with 16 maps → together represent the locations of 16 features in the input image.
- **These feature maps contain rich detail about what is in an image and where it is.**
- Faster R-CNN takes advantage of this rich detail to propose ROI locations.
  - **This enables a CNN to seamlessly perform all three steps of the object-detection process → unified model architecture.**
- It builds on R-CNN and Fast R-CNN but is markedly quicker.

YOLO

- Within each of the last models described thus far, the CNN focused on the individual proposed ROIs as opposed to the whole input image.
- Joseph Redmon and coworkers published on You Only Look Once (YOLO) in 2015, which bucked this trend.
- YOLO begins with a **pretrained** CNN for feature extraction.
- Next, the image is divided into a series of cells, and, for each cell, a number of bounding boxes and object-classification probabilities are predicted.
  - Bounding boxes with class probabilities above a threshold value are selected, and these combine to locate an object within an image.

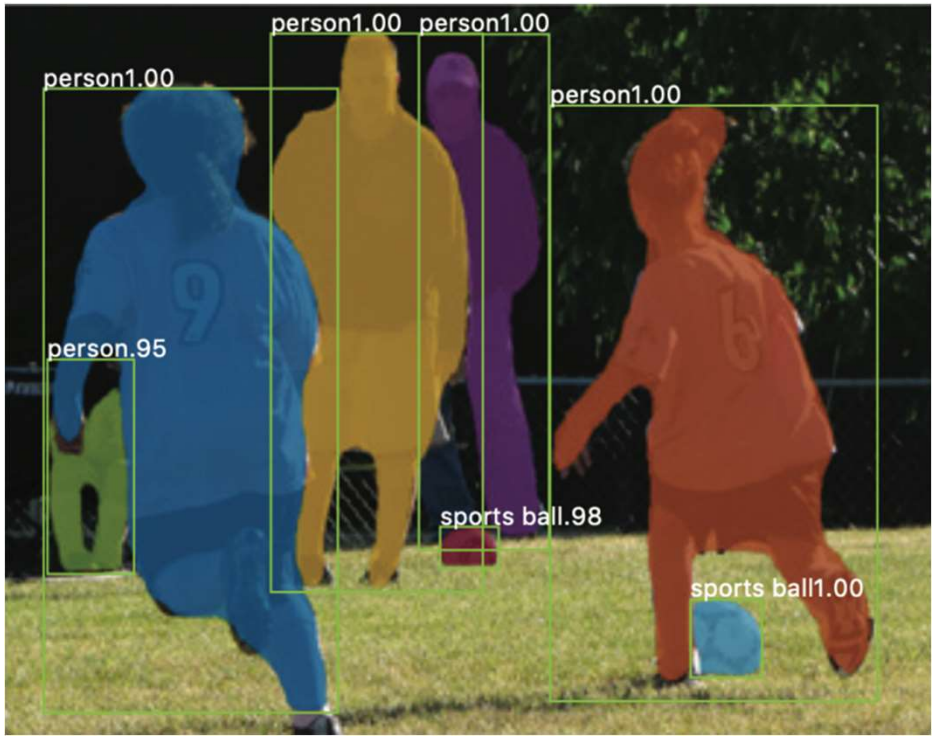
YOLO

- We can think of the YOLO method as aggregating many smaller bounding boxes, but only if they have a reasonably good probability of containing any given object class.
- The algorithm improved on the speed of Faster R-CNN, but it struggled to accurately detect small objects in an image.
- Since the original YOLO paper, Redmon and his colleagues have released their YOLO9000 and YOLOv3 models.
  - YOLO9000 resulted in increases in both execution speed and model accuracy.
  - YOLOv3 yielded some speed for even further improved accuracy - in large part due to the increased sophistication of the underlying model architectures.
- These models represent the cutting edge of object-detection algorithms.

Image Segmentation

- When the visual field of a human is exposed to a real-world scene containing many overlapping visual elements, the adult brain seems to effortlessly distinguish figures from the background.
- It defines the boundaries of these figures and relationships between them within a few hundred milliseconds.
- We will now cover **image segmentation**, another application area where deep learning has in a few short years bridged much of the gap in visual capability between humans and machines.
- We focus on two prominent model architectures - Mask R-CNN and U-Net - that are able to reliably classify objects in an image on a pixelwise scale.

Image Segmentation



This is an example of image segmentation (as performed by the Mask R-CNN algorithm). Whereas object detection involves defining object locations with coarse bounding boxes, image segmentation predicts the location of objections to the pixel level.

Mask R-CNN

Using the existing Faster R-CNN architecture to propose ROIs within the image

that are likely to contain objects.

2. An ROI classifier predicting what kind of object exists in the bounding box while

also refining the location and size of the bounding box.

3. Using the bounding box to grab the parts of the feature maps from the underlying

CNN that correspond to that part of the image.

4. Feeding the feature maps for each ROI into a fully convolutional network that

outputs a mask indicating which pixels correspond to the object in the image. An

example of such a mask—consisting of bright colors to designate the pixels associated

with separate objects—is provided in Figure 10.15.

Mask R-CNN

- Image segmentation problems require **binary masks** as labels for training.
  - These consist of arrays of the same dimensions as the original image.
- Instead of RGB pixel values they contain 1s and 0s indicating where in the image the object is:
  - 1s representing a given object's pixel-by-pixel location.
  - 0s representing everywhere else.
- Image contains a dozen different images → it must have a dozen binary masks.

U-Net

- Another popular image segmentation model is U-Net, which was developed at the University of Freiberg.

- U-Net was created for the purpose of segmenting biomedical images.
  - It outperformed the best available methods in two challenges held by the International Symposium on Biomedical Images.

- The model consists of a fully convolutional architecture:

1) It begins with a **contracting path** that produces successively smaller and deeper activation maps through multiple convolution and max-pooling steps.

1) Subsequently, an expanding path restores these deep activation maps back to full resolution through multiple upsampling and convolution steps.

U-Net

- **These two paths (contracting and expanding) are symmetrical: forming a “U” shape.**
- This symmetry → activation maps from the contracting path can be concatenated onto these expanding paths.
- Contracting paths → allow the model to learn high-resolution features from the image.
  - These features are handed directly to the expanding path.
- By the end of the expanding path → we expect the model to have localized these features within the final image dimensions.

U-Net

- Ater concatenating the feature maps from the contracting path onto the expanding path:
  - a subsequent convolutional layer allows the network to learn to assemble and localize these features precisely.

- **The final result is a network that is highly adept both at identifying features and at locating those features within two-dimensional space.**

Transfer Learning

- To be effective, many of the models we describe in this chapter are trained on very large datasets of diverse images.
  - This training requires significant **compute resources**, and the **datasets themselves are not cheap or easy to assemble**.
- Over the course of this training, a given CNN learns to extract general features from the images.
  - Low level → lines, edges, colors, and simple shades.
  - Higher level → textures, combinations of shapes, parts of objects, and complex visual elements.
- CNN has been trained on a suitably varied set of images and it is sufficiently deep → these feature maps likely contain a rich library of visual elements that can be assembled and combined to form nearly any image.

Transfer Learning

- As example, a feature map that identifies:
  - A dimple texture;
  - Combined with another that recognizes round objects;
  - And yet another that responds to white colors;
  - All three could be recombined to correctly identify a golf ball.

- **Transfer learning** takes advantage of this library of existing visual elements contained within the feature maps of a pretrained CNN and repurposes them to become specialized in identifying new classes of objects.

Transfer Learning

- Say, for example, that we'd like to build a machine vision model that performs the binary classification task we've addressed before: **distinguish hot dogs x not hot dogs**.

- **Of course, we could design a large and complex CNN that takes in images of hot dogs and not hot dogs, and outputs a single sigmoid class prediction.**

- We could train this model on a large number of training images.
  - We'd expect the convolutional layers early in the network to learn a set of feature maps that will identify hot dog-like features.

- This would work pretty well.

Transfer Learning

- However, we would need:
  - A lot of time.
  - A lot of compute power to train the CNN properly.
  - A large number of diverse images so that the CNN could learn a suitably diverse set of feature maps.

- **Transfer learning: instead of training a model from scratch, you can leverage the power of a deep model that has already been trained on a large set of images and quickly repurpose it to detecting hot dogs specifically.**

Transfer Learning

- We saw in class that VGGNet is as an example of a classic machine vision model architecture.
- In our VGGNet in Keras Jupyter notebook, we showcase the VGGNet16 model, which is composed of 16 layers of artificial neurons (conv-pool blocks).
- The closely related VGGNet19 model, which incorporates one further conv-pool block (containing three convolutional layers), is our pick for our transfer-learning starting point.
- Let's check the *Transfer Learning in Keras* notebook, in which we load VGGNet19 and modify it for our own hot-doggy purposes

Transfer Learning

- Handily, Keras provides the network architecture and parameters (called weights, but includes biases, too) already, so loading the pretrained model is easy.
  - For other pretrained Keras models, including the ResNet: keras.io/applications.
- Arguments passed to the VGG19 function help to define some characteristics of the loaded model.
- `include_top=False` specifies that we do not want the final dense classification layers from the original VGGNet19 architecture.
  - These layers were trained for classifying the original ImageNet data.
  - Rather, we'll make our own top layers and train them ourselves using our own data.

Transfer Learning

- `weights='imagenet'` is to load model parameters trained on the 14 million-sample ImageNet dataset.
- `input_shape=(224,224,3)` initializes the model with the correct input image size to handle our hot dog data.
- After we load the model, a quick `for` loop traverses each layer in the model and sets its `trainable` flag to `False` so that the parameters in these layers will not be updated during training.
- We are confident that the convolutional layers of VGGNet19 have been effectively trained to represent the generalized visual-imagery features of the large ImageNet dataset, so we leave the base model intact.

Capsule Networks

- In 2017, Sara Sabour and her colleagues on Geoff Hinton's Google Brain team in Toronto created a novel concept called **capsule networks**.
- Capsule networks have received considerable interest, because they are able to take **positional information** into consideration.
- CNNs, to their great detriment, do not.
  - A CNN would, for example, consider both of the images in the next figure to be a human face.

Capsule Networks

- The theory behind capsule networks is beyond the scope of this class, but machine vision practitioners are generally aware of them.
  - So it's good to be sure we were.
- Capsule networks have received considerable interest, because they are able to take **positional information** into consideration.
- Today, they are **too computationally intensive to be predominant in applications**, **but cheaper compute and theoretical advancements could mean that this situation will change soon**.

Capsule Networks

With convolutional neural networks, which are agnostic to the relative positioning of image features, the figure on the left and the one on the right are equally likely to be classified as Geoff Hinton's face. Capsule networks, in contrast, take positional information into consideration, and so would be less likely to mistake the right-hand figure for a face.

Referências

1. Capítulo 10 do livro *“Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence”* de Jon Krohn, Grant Beyleveld e Aglaé Bassens.