THE UNIVERSITY OF CHICAGO

Machine Learning and Predictive Analytics, MSCA-31009-4

Lecture 7

Igor Yakushin

May 18, 2021

Overview I

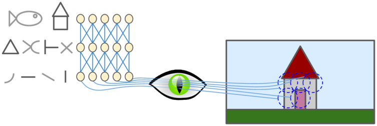
1. Convolutional Neural Networks
    - The Architecture of the Visual Cortex
    - Convolutional layers
        - Filters
        - Stacking Multiple Feature Maps
    - Pooling Layers
    - CNN Architectures
        - LeNet-5
        - AlexNet
        - GoogLeNet
        - VGGNet
        - ResNet
        - Xception
        - SENet
    - Using Pretrained Models from Keras
    - Pretrained Models for Transfer Learning

Convolutional Neural Networks

- Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s.
- In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in the previous lecture 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.
- Moreover, CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing.
- However, we will focus on visual applications for now.

The Architecture of the Visual Cortex

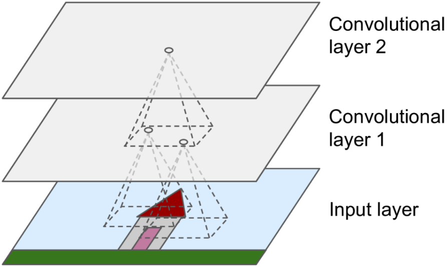
- From experiments on animals it was noticed that many neurons in the visual cortex have a small local receptive field, meaning they react only to visual stimuli located in a limited region of the visual field
- The receptive fields of different neurons may overlap, and together they tile the whole visual field.
- Some neurons react only to images of horizontal lines, while others react only to lines with different orientations
- Two neurons may have the same receptive field but react to different line orientations
- Some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns



- These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons

Convolutional layers

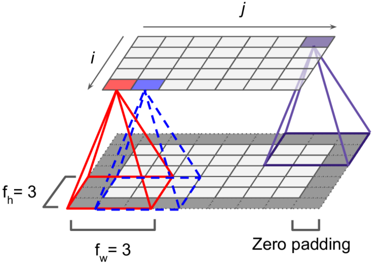
- The most important building block of a CNN is the convolutional layer
- Neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in the dense layers before), but only to pixels in their receptive fields.
- In turn, each neuron in the second 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.

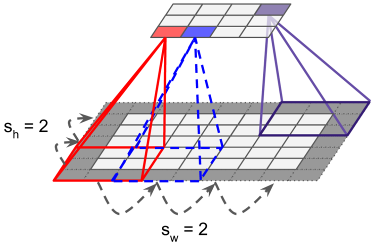
Convolutional layers

- All the multilayer neural networks we've looked at so far had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. In a CNN each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.
- A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h – 1$, columns j to $j + f_w – 1$, where f_h and f_w are the height and width of the receptive field



- 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, as shown in the diagram. This is called zero padding.

Convolutional layers

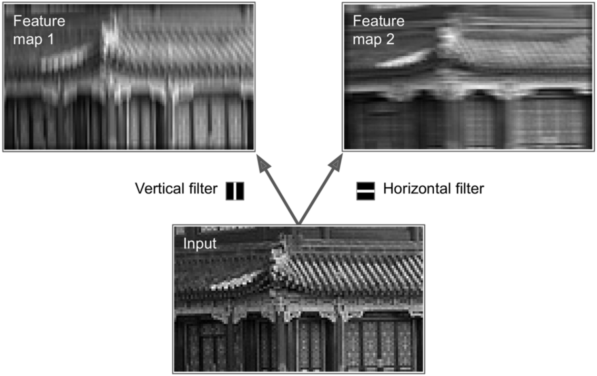


- It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields
- The shift from one receptive field to the next is called the stride.

- A 5 × 7 input layer (plus zero padding) is connected to a 3 × 4 layer, using 3 × 3 receptive fields and a stride of 2
- In general, the stride can be different in different directions
- A neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h{-}1$, columns $j \times s_w$ to $j \times s_w + f_w{-}1$, where s_h and s_w are the vertical and horizontal strides

Filters

- A neuron's weights can be represented as a small image the size of the receptive field.
- For example, the picture shows two sets of weights, called filters or convolution kernels



Filters

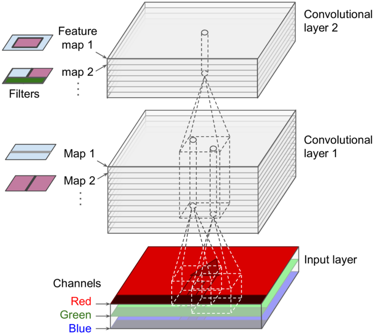
- The first one is represented as a black square with a vertical white line in the middle
- It is a 7×7 matrix full of 0s except for the central column, which is full of 1s
- The neurons using these weights will ignore everything in their receptive field except for the central vertical line since all inputs will get multiplied by 0, except for the ones located in the central vertical line.
- The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.
- Now if all neurons in a layer use the same vertical line filter and the same bias term, and you feed the network the input image shown in the bottom, the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred.

Filters

- 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.
- Thus, a layer full of neurons using the same filter outputs a feature map, which highlights the areas in an image that activate the filter the most.
- Of course, you do not have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

Stacking Multiple Feature Maps

- A convolutional layer has multiple filters - you decide how many - and outputs one feature map per filter, so it is more accurately represented in 3D



- There is one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (weights and bias).
- Neurons in different feature maps use different parameters
- A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps.

- A convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

Stacking Multiple Feature Maps

- 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.
- In contrast, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location.
- Input images are also 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 much more - for example, satellite images that capture extra light frequencies (such as infrared).

Stacking Multiple Feature Maps

- A neuron located in row i, column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer l–1, located in rows $i \times s_h$ to $i \times s_h + f_h$–1 and columns $j \times s_w$ to $j \times s_w + f_w$–1, across all feature maps (in layer l–1). Note that all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k}$$

$$i' = i \times s_h + u$$

$$j' = j \times s_w + v$$

Stacking Multiple Feature Maps

- $z_{i,j,k}$ - output of the neuron in row i, column j in feature map k of the convolutional layer l
- $x_{i',j',k'}$ - output of the neuron in layer $l - 1$
- b_k - bias term for feature map k in layer l. Controls the overall brightness of the feature map.
- $w_{u,v,k',k}$ - connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'

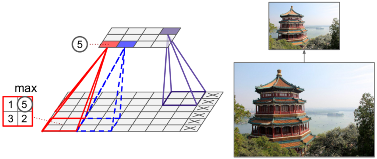
In Keras:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3,
                           strides=1, padding="same",
                           activation="relu")
```

Cells 3-13, l7_n1.ipynb

Pooling Layers

- The second common building block of CNNs: the pooling layer
- Their goal of a pooling layer is to subsample the image to reduce the computational load, the memory usage, and the number of parameters
- Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean.

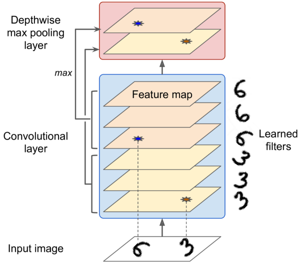


- Max pooling layer is the most common type of pooling layer.
- In this example: 2×2 pooling kernel, with a stride of 2 and no padding. Only the max input value in each receptive field makes it to the next layer

Pooling Layers

In Keras:

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```



- Note that max pooling and average pooling can be performed along the depth dimension rather than the spatial dimensions, although this is not as common
- This can allow the CNN to learn to be invariant to various features.
- For example, it could learn multiple filters, each detecting a different rotation of the same pattern (such as handwritten digits), and the depthwise max pooling layer would ensure that the output is the same regardless of the rotation.
- The CNN could similarly learn to be invariant to anything else: thickness, brightness, skew, color, and so on.

Pooling Layers

- Keras does not have depthwise pooling but one can wrap low level TensorFlow call:

```
depth_pool = keras.layers.Lambda(lambda X:
               tf.nn.max_pool(X, ksize=(1, 1, 1, 3),
                             strides=(1, 1, 1, 3),
                             padding="valid"))
```

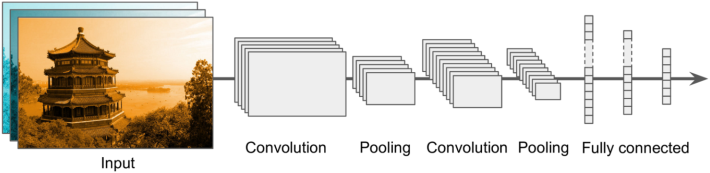
- One last type of pooling layer that you will often see in modern architectures is the global average pooling layer.
- It computes the mean of each entire feature map
- It can sometimes be useful as an output layer

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

- Cells 14-25, l7_n1.ipynb

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 - more feature maps - thanks to the convolutional layers.
- 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 (for example, a softmax layer that outputs estimated class probabilities)



CNN Architectures

Here is a simple CNN that can be used for the Fashion MNIST dataset

```
model = keras.models.Sequential([
keras.layers.Conv2D(64, 7, activation="relu", padding="same",
input_shape=[28, 28, 1]),
keras.layers.MaxPooling2D(2),
keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
keras.layers.MaxPooling2D(2),
keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
keras.layers.MaxPooling2D(2),
keras.layers.Flatten(),
keras.layers.Dense(128, activation="relu"),
keras.layers.Dropout(0.5),
keras.layers.Dense(64, activation="relu"),
keras.layers.Dropout(0.5),
keras.layers.Dense(10, activation="softmax")
])
```

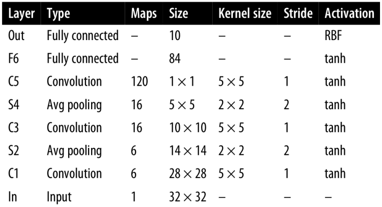
- Note that the number of filters grows as we climb up the CNN toward the output layer (it is initially 64, then 128, then 256): it makes sense for it to grow, since the number of low-level features is often fairly low (e.g., small circles, horizontal lines), but there are many different ways to combine them into higher-level features.

CNN Architectures

- It is a common practice to double the number of filters after each pooling layer: since a pooling layer divides each spatial dimension by a factor of 2, we can afford to double the number of feature maps in the next layer without fear of exploding the number of parameters, memory usage, or computational load.
- On the top there is a fully connected network, composed of two hidden dense layers and a dense output layer. Note that we must flatten its inputs, since a dense network expects a 1D array of features for each instance.
- We also add two dropout layers, with a dropout rate of 50% each, to reduce overfitting
- This CNN reaches over 92% accuracy on the test set. It's not state of the art, but it is pretty good, and clearly much better than what we achieved with dense networks. Cells 26-29, l7_n1.ipynb

LeNet-5

- The LeNet-5 was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition by the post office.



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	–	–	–

- The average pooling layers are slightly more complex than usual: each neuron computes the mean of its inputs, then multiplies the result by a learnable coefficient (one per map) and adds a learnable bias term (again, one per map), then finally applies the activation function.

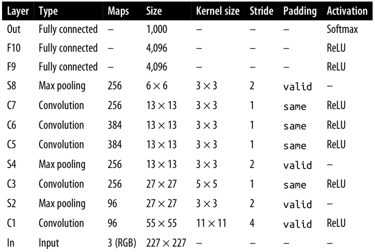
- Most neurons in C3 maps are connected to neurons in only three or four S2 maps (instead of all six S2 maps)

LeNet-5

- The output layer is a bit special: instead of computing the matrix multiplication of the inputs and the weight vector, each neuron outputs the square of the Euclidian distance between its input vector and its weight vector. Each output measures how much the image belongs to a particular digit class. The cross-entropy cost function is now preferred, as it penalizes bad predictions much more, pro- ducing larger gradients and converging faster

AlexNet

- The AlexNet CNN architecture 11 won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best achieved only 26%
- It was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. It is similar to LeNet-5, only much larger and deeper



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	–	–	–	–

- To reduce overfitting, the authors used two regularization techniques: dropout with a 50% dropout rate applied to the outputs of layers F9 and F10; data augmentation by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

AlexNet

- AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called local response normalization (LRN): the most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps (such competitive activation has been observed in biological neurons).
- This encourages different feature maps to specialize, pushing them apart and forcing them to explore a wider range of features, ultimately improving generalization.

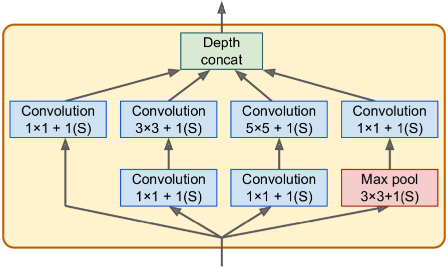
$$\begin{aligned} b_i &= a_i\Big(k + \alpha \sum_{j=j_{low}}^{j_{high}} a_j^2\Big)^{-\beta} \\ j_{high} &= \min(i + \frac{r}{2}, f_n - 1) \\ j_{low} &= \max(0, i - \frac{r}{2}) \end{aligned} \tag{1}$$

AlexNet

- b_i is the normalized output of the neuron located in feature map i, at some row u and column v
- a_i is the activation of that neuron after the ReLU step, but before normalization
- k, α, β, and r are hyperparameters. k is called the bias, and r is called the depth radius.
- f_n is the number of feature maps.
- In AlexNet, the hyperparameters are set as follows: $r = 2$, $\alpha = 0.00002$, $\beta = 0.75$, and $k = 1$. This step can be implemented using the tf.nn.local_response_normalization() function which you can wrap in a Lambda layer if you want to use it in a Keras model.
- A variant of AlexNet called ZF Net 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.).

GoogLeNet

- The GoogLeNet architecture was developed by Christian Szegedy and others 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).

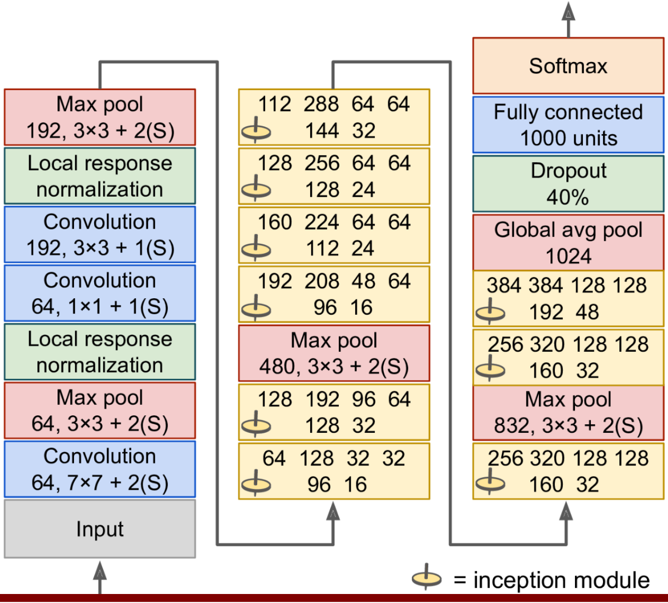
GoogLeNet

- The notation $3 \times 3 + 1(S)$ means that the layer uses a 3×3 kernel, stride 1, and ”same” padding.
- The input signal is first copied and fed to four different layers. All convolutional layers use the ReLU activation function.
- Note that the second set of convolutional layers uses different kernel sizes (1×1, 3×3, and 5×5), allowing them to capture patterns at different scales.
- Also note that 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.
- This makes it possible to concatenate all the outputs along the depth dimension in the final depth concatenation layer (i.e., stack the feature maps from all four top convolutional layers). This concatenation layer can be implemented in TensorFlow using the tf.concat() operation, with axis=3 (the axis is the depth).

GoogLeNet

- Why 1×1 kernels are used? They look at only one pixel at a time.
  - Although they cannot capture spatial patterns, they can capture patterns along the depth dimension.
  - They are configured to output fewer feature maps than their inputs, so they serve as bottleneck layers, meaning they reduce dimensionality. This cuts the computational cost and the number of parameters, speeding up training and improving generalization.
  - Each pair of convolutional layers ($[1 \times 1, 3 \times 3]$ and $[1 \times 1, 5 \times 5]$) acts like a single powerful convolutional layer, capable of capturing more complex patterns. Indeed, instead of sweeping a simple linear classifier across the image (as a single convolutional layer does), this pair of convolutional layers sweeps a two-layer neural network across the image.
- You can think of the whole inception module as a convolutional layer on steroids, able to output feature maps that capture complex patterns at various scales.

GoogLeNet


Input
Convolution
64, 7×7 + 2(S)
Max pool
64, 3×3 + 2(S)
Local response
normalization
Convolution
64, 1×1 + 1(S)
Convolution
192, 3×3 + 1(S)
Local response
normalization
Max pool
192, 3×3 + 2(S)
64 128 32 32
96 16
128 192 96 64
128 32
Max pool
480, 3×3 + 2(S)
192 208 48 64
96 16
160 224 64 64
112 24
128 256 64 64
128 24
112 288 64 64
144 32
256 320 128 128
160 32
Max pool
832, 3×3 + 2(S)
256 320 128 128
160 32
384 384 128 128
192 48
Global avg pool
1024
Dropout
40%
Fully connected
1000 units
Softmax
= inception module

GoogLeNet

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load. The first layer uses a large kernel size so that much of the information is preserved.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features (as discussed earlier).
- Two convolutional layers follow, where the first acts like a bottleneck layer. You can think of this pair as a single smarter convolutional layer.
- Again, a local response normalization layer ensures that the previous layers capture a wide variety of patterns.
- Next, a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the tall stack of nine inception modules, interleaved with a couple max pooling layers to reduce dimensionality and speed up the net.

GoogLeNet

- Next, the global average pooling layer outputs the mean of each feature map: this drops any remaining spatial information, which is fine because there was not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be 224×224 pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to 7×7. Moreover, it is a classification task, not localization, so it does not matter where the object is. Thanks to the dimensionality reduction brought by this layer, there is no need to have several fully connected layers at the top of the CNN (like in AlexNet), and this considerably reduces the number of parameters in the network and limits the risk of overfitting.
- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with 1,000 units (since there are 1,000 classes) and a softmax activation function to output estimated class probabilities.

GoogLeNet

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

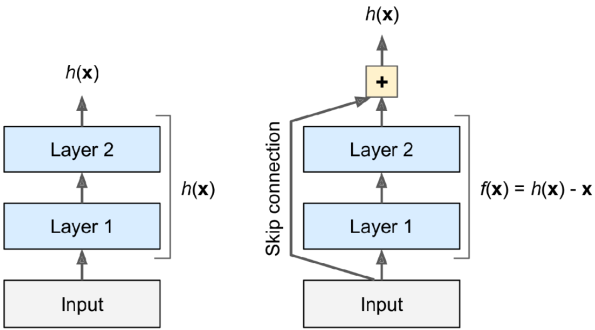
VGGNet

- The runner-up in the ILSVRC 2014 challenge was VGGNet, developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG) research lab at Oxford University.
- It had a very simple and classical architecture, with 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 just 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 only 3×3 filters, but many filters.

ResNet

- Kaiming He et al. won the ILSVRC 2015 challenge using a Residual Network (or 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: models are 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 a bit higher up the stack.
- When training a neural network, the goal is to make it model a target function $h(x)$. 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 $f(x) = h(x)–x$ rather than $h(x)$. This is called residual learning.

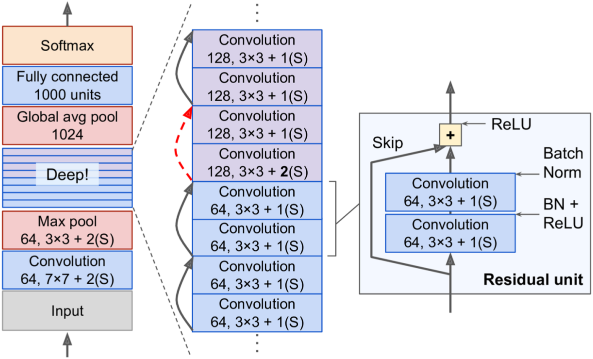
ResNet



- When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero.
- If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function.
- If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

ResNet

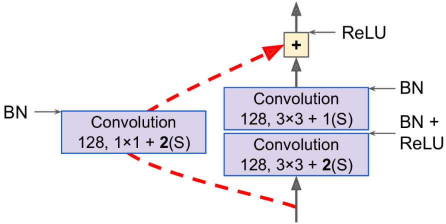
- Moreover, if you add many skip connections, the network can start making progress even if several layers have not started learning yet. Thanks to skip connections, the signal can easily make its way across the whole network. The deep residual network can be seen as a stack of residual units (RUs), where each residual unit is a small neural network with a skip connection.



Softmax
Fully connected 1000 units
Global avg pool 1024
Deep!
Max pool 64, 3×3 + 2(S)
Convolution 64, 7×7 + 2(S)
Input
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 1(S)
Convolution 128, 3×3 + 2(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
Convolution 64, 3×3 + 1(S)
ReLU
Skip
+
Batch Norm
Convolution 64, 3×3 + 1(S)
BN + ReLU
Convolution 64, 3×3 + 1(S)
Residual unit

ResNet

- Note that the number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2). When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow).
- To solve this problem, the inputs are passed through a 1×1 convolutional layer with stride 2 and the right number of output feature maps.



ResNet

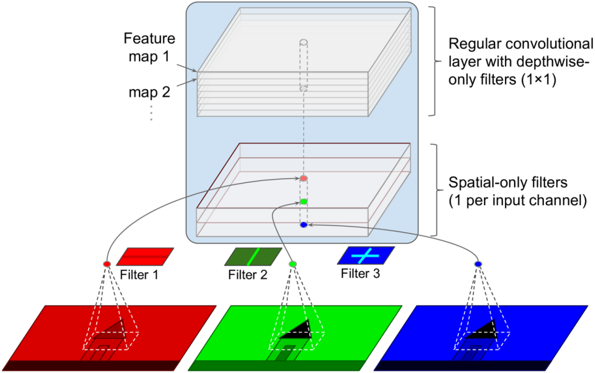
- ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer) containing 3 residual units that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps. Cells 30-32, l7_n1.ipynb
- ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two 3×3 convolutional layers with, say, 256 feature maps, they use three convolutional layers: first a 1×1 convolutional layer with just 64 feature maps (4 times less), which acts as a bottleneck layer, then a 3×3 layer with 64 feature maps, and finally another 1×1 convolutional layer with 256 feature maps (4 times 64) that restores the original depth. ResNet-152 contains 3 such RUs that output 256 maps, then 8 RUs with 512 maps, a whopping 36 RUs with 1,024 maps, and finally 3 RUs with 2,048 maps.
- Inception-v4 architecture merged the ideas of GoogLeNet and ResNet and achieved an error rate of close to 3% on ImageNet classification.

Xception

- Xception - Extreme Inception - is another variant of the GoogLeNet architecture. It 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 (or separable convolution layer for short 20).
- While a regular convolutional layer uses filters that try to simultaneously capture spatial patterns (e.g., an oval) and cross-channel patterns (e.g., mouth + nose + eyes = face), a separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately

Xception

- Thus, it is composed of two parts: the first part applies a single spatial filter for 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

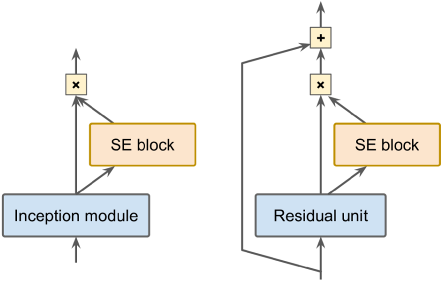

Feature map 1
map 2
Regular convolutional layer with depthwise-only filters (1×1)
Spatial-only filters (1 per input channel)
Filter 1
Filter 2
Filter 3

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).
- Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and in general they even perform better, so you should consider using them by default (except after layers with few channels).

SENet

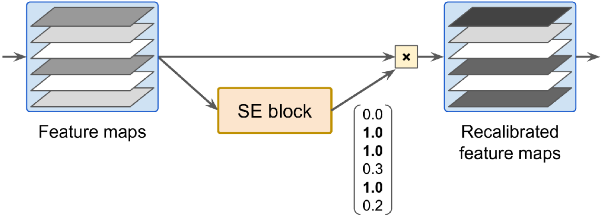
- The winning architecture in the ILSVRC 2017 challenge was the Squeeze-and-Excitation Network - SENet.
- This architecture extends existing architectures such as inception networks and ResNets, and boosts their performance. This allowed SENet to win the competition with 2.25% top-five error rate!
- The extended versions of inception networks and ResNets are called SE-Inception and SE-ResNet, respectively.



- The boost comes from the fact that a SENet adds a small neural network, called an SE block, to every unit in the original architecture (i.e., every inception module or every residual unit)

SENet

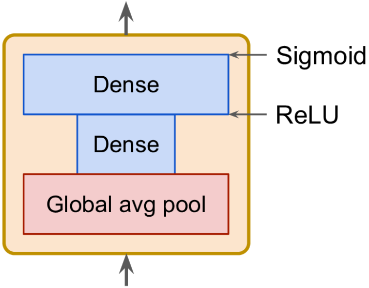
- An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together. It then uses this information to recalibrate the feature maps



- For example, an SE block may learn that mouths, noses, and eyes usually appear together in pictures: if you see a mouth and a nose, you should expect to see eyes as well. So if the block sees a strong activation in the mouth and nose feature maps, but only mild activation in the eye feature map, it will boost the eye feature map

SENet

- An SE block is composed of just three layers: a global average pooling layer, a hidden dense layer using the ReLU activation function, and a dense output layer using the sigmoid activation function



SENet

- The global average pooling layer computes the mean activation for each feature map: for example, if its input contains 256 feature maps, it will output 256 numbers representing the overall level of response for each filter.
- The next layer is where the “squeeze” happens: this layer has significantly fewer than 256 neurons - typically 16 times fewer than the number of feature maps - so the 256 numbers get compressed into a small vector . This is a low- dimensional vector representation (i.e., an embedding) of the distribution of feature responses. This bottleneck step forces the SE block to learn a general representation of the feature combinations
- Finally, the output layer takes the embedding and outputs a recalibration vector containing one number per feature map (e.g., 256), each between 0 and 1. The feature maps are then multiplied by this recalibration vector, so irrelevant features get scaled down while relevant features are left alone

Using Pretrained Models from Keras

- In general, you won't have to implement standard models like GoogLeNet or ResNet manually, since pretrained networks are readily available with a single line of code in the keras.applications package:

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

- To use it, you first need to ensure that the images have the right size. A ResNet-50 model expects 224 × 224-pixel images (other models may expect other sizes, such as 299 × 299):

```
images_resized = tf.image.resize(images, [224, 224])
```

- The tf.image.resize() will not preserve the aspect ratio. If this is a problem, try cropping the images to the appropriate aspect ratio before resizing. Both operations can be done in one shot with tf.image.crop_and_resize()

Using Pretrained Models from Keras

- The pretrained models assume that the images are preprocessed in a specific way. In some cases they may expect the inputs to be scaled from 0 to 1, or −1 to 1, and so on.
- Each model provides a preprocess_input() function that you can use to preprocess your images. These functions assume that the pixel values range from 0 to 255

```
inputs =
 keras.applications.resnet50.preprocess_input(images_resized)
```

- Now we can use the pretrained model to make predictions:

```
Y_proba = model.predict(inputs)
```

Using Pretrained Models from Keras

- The output Y_proba is a matrix with one row per image and one column per class (in this case, there are 1,000 classes).
- If you want to display the top K predictions, including the class name and the estimated probability of each predicted class, use the decode_predictions() function. For each image, it returns an array containing the top K predictions, where each prediction is represented as an array containing the class identifier, its name, and the corresponding confidence score:

```
top_K =
  keras.applications.resnet50.decode_predictions(Y_proba,
                                                 top=3)
for image_index in range(len(images)):
  print("Image #{}".format(image_index))
  for class_id, name, y_proba in top_K[image_index]:
    print(" {} - {:12s} {:.2f}%".format(class_id, name,
                                        y_proba * 100))
```

Using Pretrained Models from Keras

- The output looks like this:

```
Image #0
  n03877845 - palace      42.87%
  n02825657 - bell_cote   40.57%
  n03781244 - monastery   14.56%
Image #1
  n04522168 - vase        46.83%
  n07930864 - cup         7.78%
  n11939491 - daisy       4.87%
```

Cells 33-40, l7_n1.ipynb

Pretrained Models for Transfer Learning

- If you want to build an image classifier but you do not have enough training data, then it is often a good idea to reuse the lower layers of a pretrained model
- Cells 41-56, l7_n1.ipynb