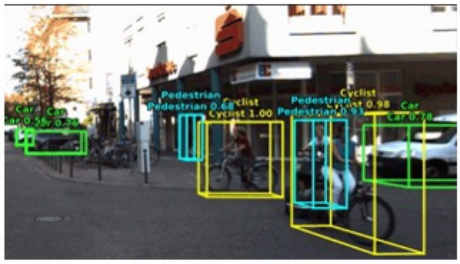
Deep Learning: Convolutional Neural Networks

Krzysztof Czarnecki
Waterloo Intelligent Systems Engineering Lab
Electrical and Computer Engineering Department

1

In this unit, I will explain convolutional neural networks, which are the main work horse in modern computer vision.

ConvNets For Self-Driving Cars



Code at: https://github.com/kujason/avod

Code at: https://github.com/oandrienko/fast-semantic-segmentation

Credit: A. Hakareh, S. Waslander

2

In automated driving, convolutional neural networks, or convnets, are indispensable for perception tasks like object detection and tracking, semantic segmentation, and behavior prediction.

Learning Objectives

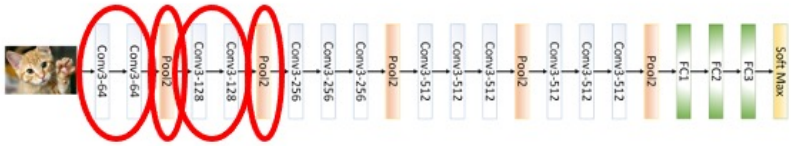
- Learn how ConvNets (aka CNNs) use cross-correlation in their hidden layers instead of general matrix multiplication.
- Learn the advantages of using ConvNets over traditional neural networks for processing images.
- Learn about modern ConvNet architectures.
- Develop an intuition of what ConvNets learn internally.

3

Partial credit: A. Hakareh, S. Waslander

ConvNets

- Used for processing data defined on N-dim grid.
- 1D time series data, 2D images, 3D videos.
- Two major type of layers:
  1. **Convolution Layers**
  2. **Pooling Layers**
- **Example: VGG 16**

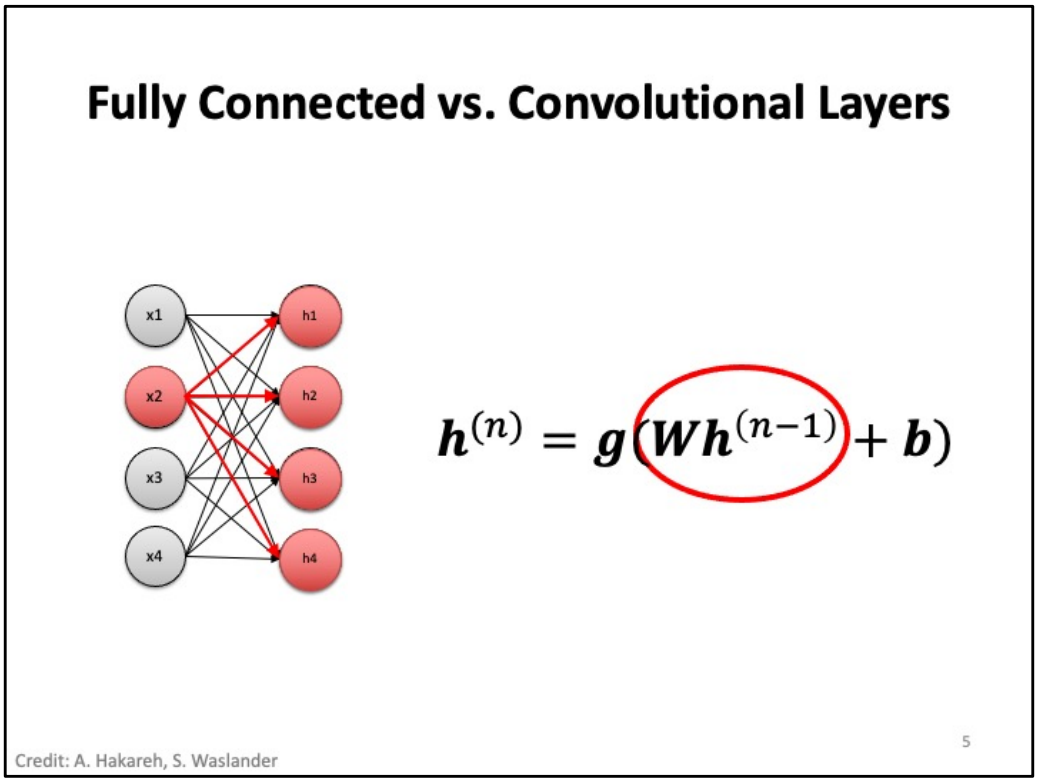


4

Credit: A. Hakareh, S. Waslander

We use convnets to process data that is defined over a n-dimensional grid. The grid can be 1-dimensional, such as for time series (like stock prices); two dimensional (like images); or 3 dimensional (like videos); some dimensions can be temporal and some spatial. For images, we have two spatial dimensions, and for videos, we have an additional temporal dimension. (However, we can also process videos using 2D convolutions by stacking the images up and treating the multiple frames like additional image channels.)

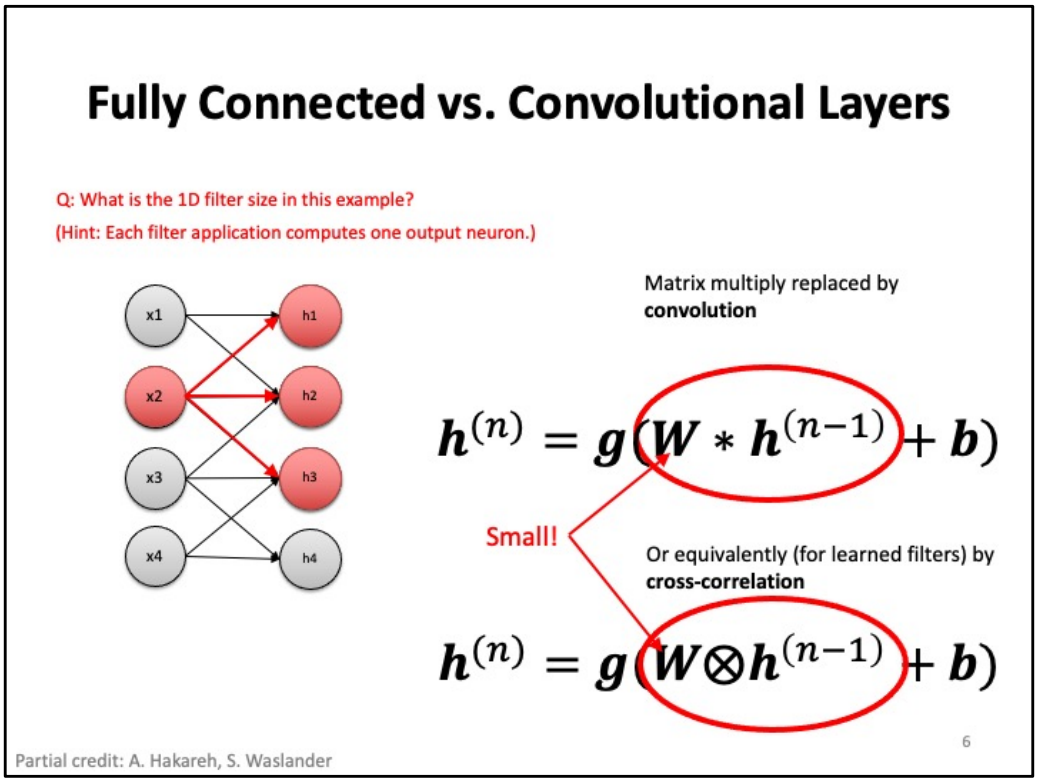
Convnets consist of two major types of layers: convolutional layers, which detect features, and pooling layers, which summarize features. For example (shown at the bottom), VGG 16 is a 16-layer convnet architecture that consists of series of intertwined convolutional and pooling layers; they may be followed by some fully connected (FC) layers and an output layer (as discussed in the previous lectures.)



Let's look at a fully connect layer, and then contrast it with a convolutional layer.

Recall that a fully connect layer is represented by a matrix-vector multiplication (circled), where every output neuron has a connection to every input neuron and vice versa. Also recall that a row in the weight matrix represents the connections from the output neuron with the same index as the row to all the input neurons. Conversely, a column represents all the connections between the input with the same index as the column and every output. The set of connections from input x2 to every output is highlighted in the diagram on the left; their weights would be the second column in $\boldsymbol{W}$.

It turns out that fully connected layers are not optimal when processing images. We usually don't want all pixels to be feeding into every output neuron; instead, we want each output neuron to focus on a specific location in the input.

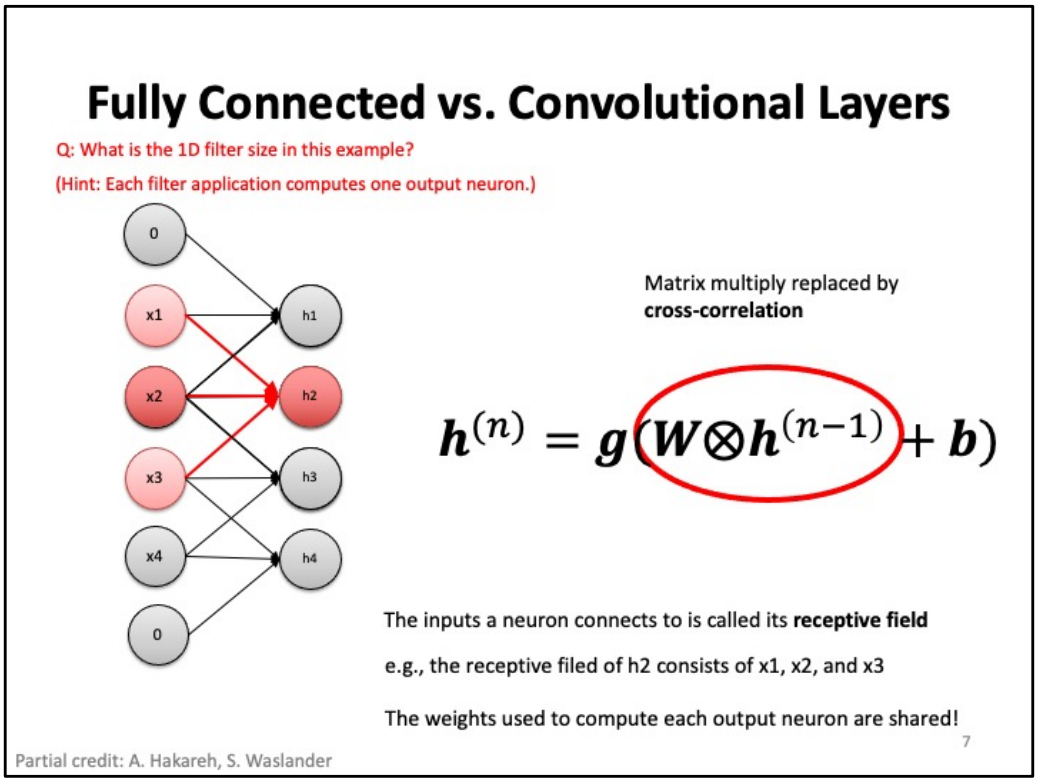


The key idea of a convolutional layer is that the matrix multiply is replace by a convolution, or equivalently, cross-correlation, and the big weight matrix (that would cover the whole image) is replaced by a small one that represents a filter, which covers one patch of the input at a time.

Whether we use convolution or cross-correlation does not matter; either way works, but most implementations use cross-correlation. If you got rusty on both operations, please review the image filtering lecture, where we discussed them in detail.

The picture on the left illustrates what happens to the layer connectivity, when we replace the full weight-matrix multiply by a convolution or cross-correlation with a small filter. Instead of having full connectivity, each input feeds only into a few neighboring output neurons. Alternatively, each output neuron, accesses only a few corresponding input neurons.

If you recall the convolution or cross-correlation operation, what is the size of the filter that is being applied in this figure? What do you think?
Here is a hint: each filter application computes one output neuron.



This diagram represents the application of a filter of size 3 in one dimension. Each output is computed from three corresponding inputs, that is, the middle one and its neighbors on each side. Since the picture on the previous slide has same number of inputs and outputs, we would also need some padding on each side (as shown in this slide).

We also say that each output neuron has a receptive filed of size 3, since it connects to three input neurons. We can also trace the receptive field of some target neuron residing in a deeper layer over its chain of direct and indirect inputs to an earlier layer, or all the way to the input layer. This “induced” receptive field of a neuron usually grows with the depth of its layer, and it is referred to as the “theoretical receptive field” (“theoretical” since the specific weights might be such that some input neurons have little impact on the target neuron).

Another key point is that each output neuron is computed using the same three weights, which is the filter that we slide over the input.

Recall: Image Filtering Using 2D Cross Correlation

$g[\cdot,\cdot]$ $\frac{1}{9}$

1	1	1
1	1	1
1	1	1

$f[.,.]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$h[.,.]$

	0	10	20	30	30	30	20	10	
	20	40	60	60	60	40	20		
	0	30	60	90	90	90	60	30	
	0	30	50	80	80	90	60	30	
	0	30	50	80	80	90	60	30	
	0	20	30	50	50	60	40	20	
	10	20	30	30	30	30	20	10	
	10	10	10	0	0	0	0	0	

$$h[m,n] = g \otimes f[m,n] = \sum_{k,l} g[k,l] f[m+k, n+l]$$

redit: S. Seitz

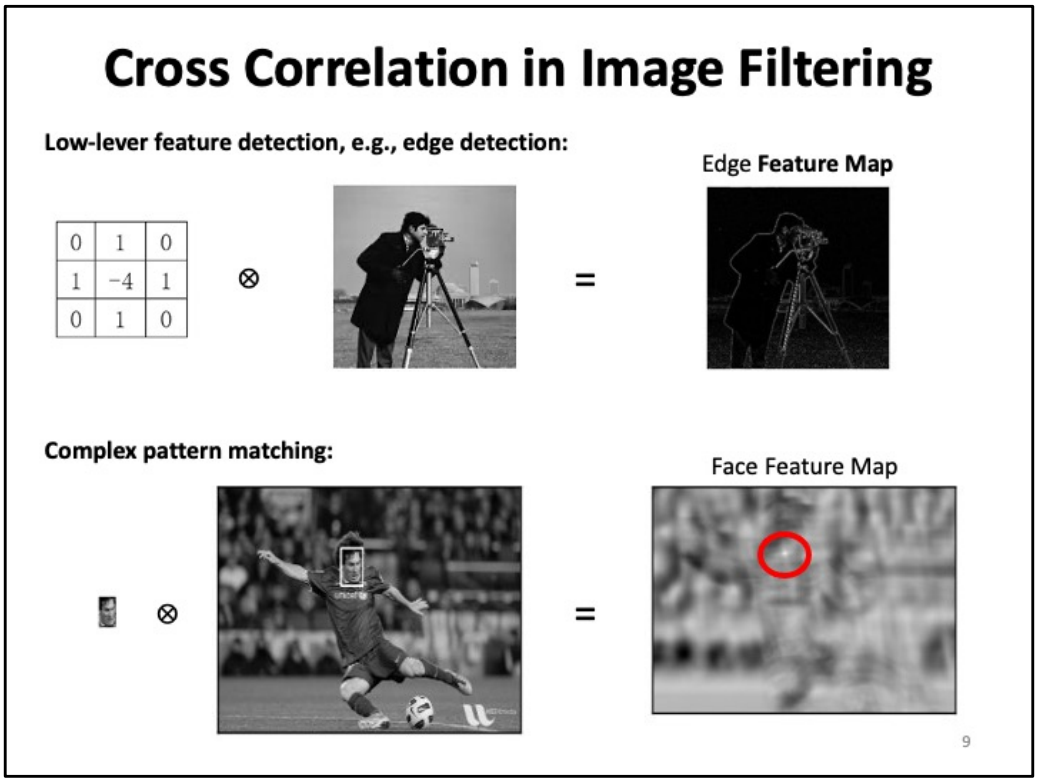
8

For image processing, we need 2D filters, and we already studied those in detail in the image filtering lecture.

Recall that the output is computed by sliding the filter g over the input f, producing a corresponding pixel in the output h. The definition below increments the indices to follow the snake pattern, but, of course, we can compute each output pixel in any order, including computing them in parallel. BTW, cross-correlation can be represented as a matrix-matrix multiplication [*]; it just needs creating a larger matrix from the filter, a so-called Toeplitz matrix. This can be exploited in the underlying implementation, especially when targeting GPUs and hardware accelerators.

---

[*] This post explains how cross-correlation can be expressed as a matrix multiply, but of different shape (i.e., involving a Toeplitz matrix): https://stackoverflow.com/questions/16798888/2-d-convolution-as-a-matrix-matrix-multiplication



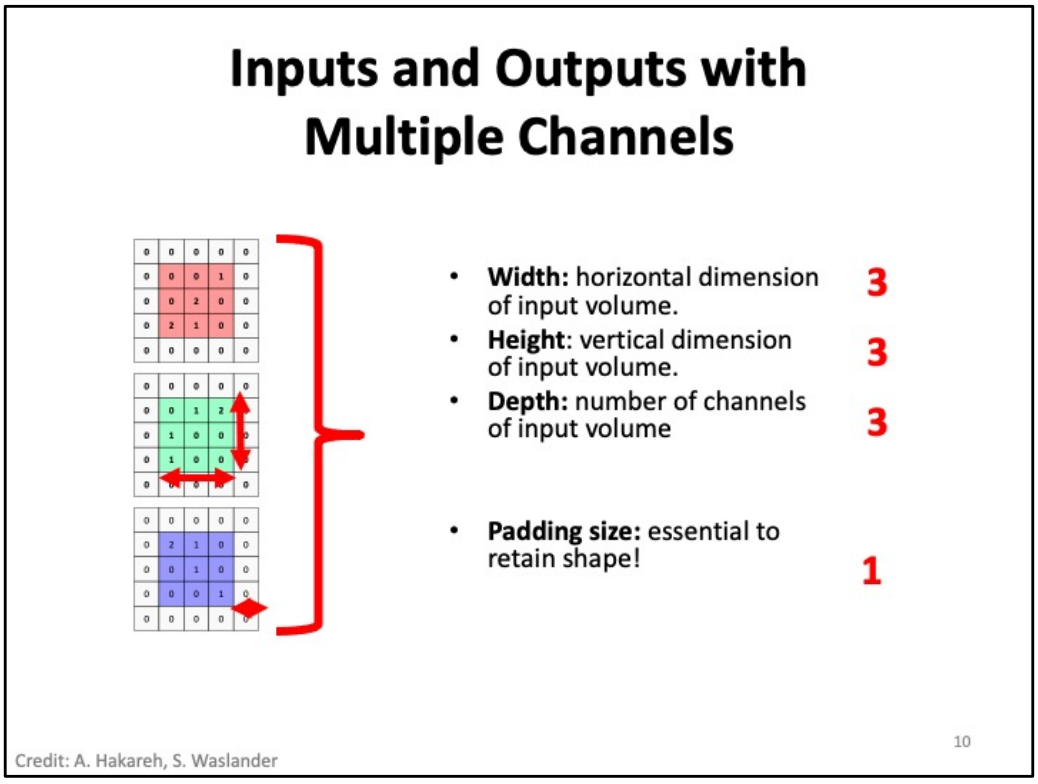
Recall that cross-correlation (and convolution) has proved to be extremely useful and versatile in computer vision, allowing us to compute low-level features, like edges. For example, here we apply a Laplace filter to the input image to produce an edge image (images at the top half). The value of each pixel in this image tells us whether there is an edge at that location in the input image. We refer to such an image as a feature map.

But we also can use cross-correlation to identify more complex patterns in the input image. For example, if we use a face as a filter, the resulting feature map will tell us all the location where the pattern matches in the input image (images at the bottom). This example uses the normalized cross-correlation: cv.TM_CCORR_NORMED in OpenCV. We see clearly see a peak (in the red circle) in the right figure, which can be thought of as the feature map for the face.

Convolutional layers perform exactly such tasks, but we let training (gradient descent) discover the filters rather than handcrafting them. Also, the filter application is followed by non-linearity to focus on strong responses. For example, a ReLU and a bias can express a threshold to suppress all the weak matches, and only pass the strong one (like the circled one).

Note that we would not let a single convolutional filter match such a big pattern like a face at once, but decompose this problem into a hierarchy of filters, where filters in

early layers recognize low-level features like edges and corners, and later layers focus on high level features like eyes, nose, and face. Such a hierarchical arrangement allows these features to be composed to cover an exponential number of combinations.

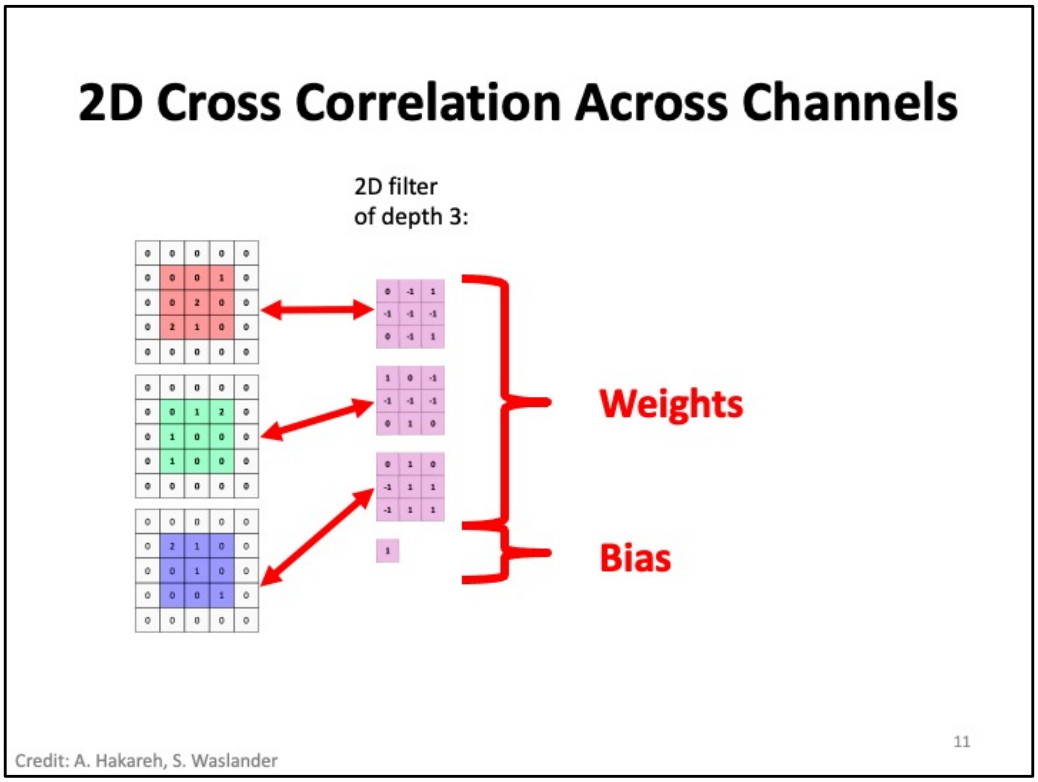


The inputs that we want to feed into convolutional layers and the outputs that we want produce from convolutional layers are multi-channel. For example, color images have RGB planes, and feature representations have multiple feature maps, such as edge intensity and edge orientation for edges. These multi-channel inputs and outputs are also called volumes (since they are 3D) or, more generally, tensors (which can have more dimensions).

We will use the following size parameters to describe such volumes:

- Width - horizontal dimension of input volume; 3 in our example on the left.
- Height - vertical dimension of input volume; also 3 in this example.
- Depth - number of channels of input volume; also 3 in this example.

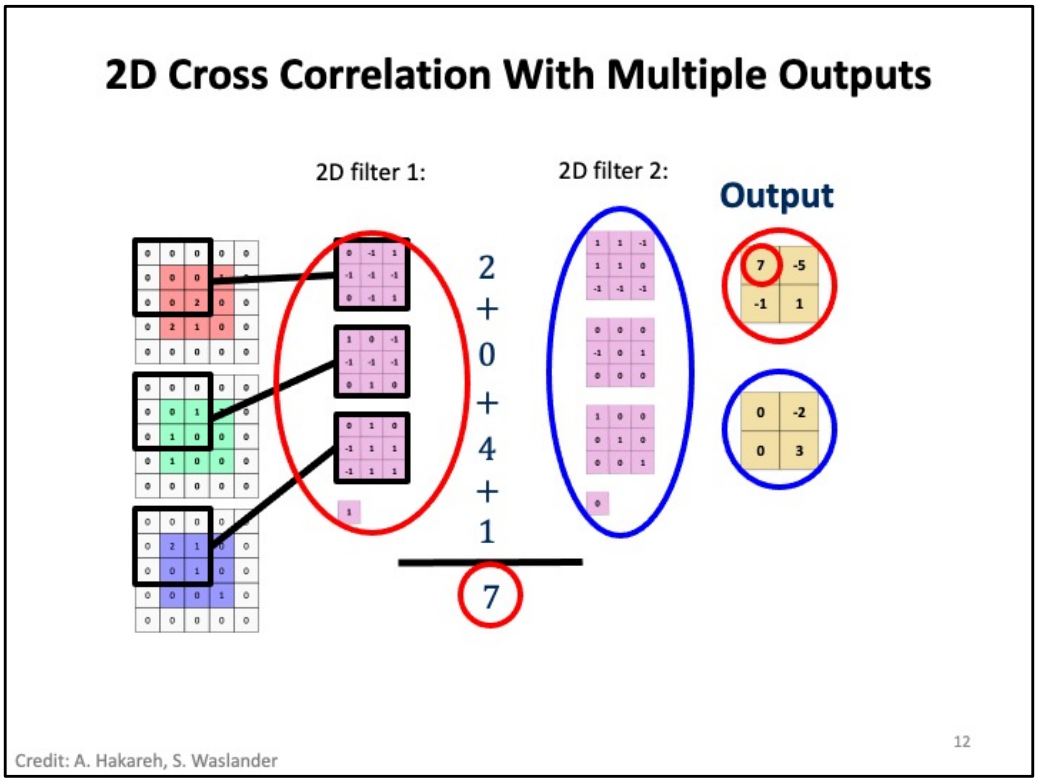
Finally, if we want to keep the width and height of the output from the conv layer same as the input, we also need to add padding. We will discuss later how to determine how much padding to add. This example uses padding of 1 (i.e., 1-pixel-wide border around the original width and height shape).



The filter that we apply to a volume also needs to have multiple channels, one for each input channel.

Therefore, the depth of the filter matches the depth of the input. The width and height of the filter is a design choice; we very often use small filters, such as 3x3, just like the many filters we saw in the filtering lecture.

We also need a bias value, which will determine the threshold for the filter response to pass through the non-linearity (e.g., a ReLU), which normally follows the filter application.

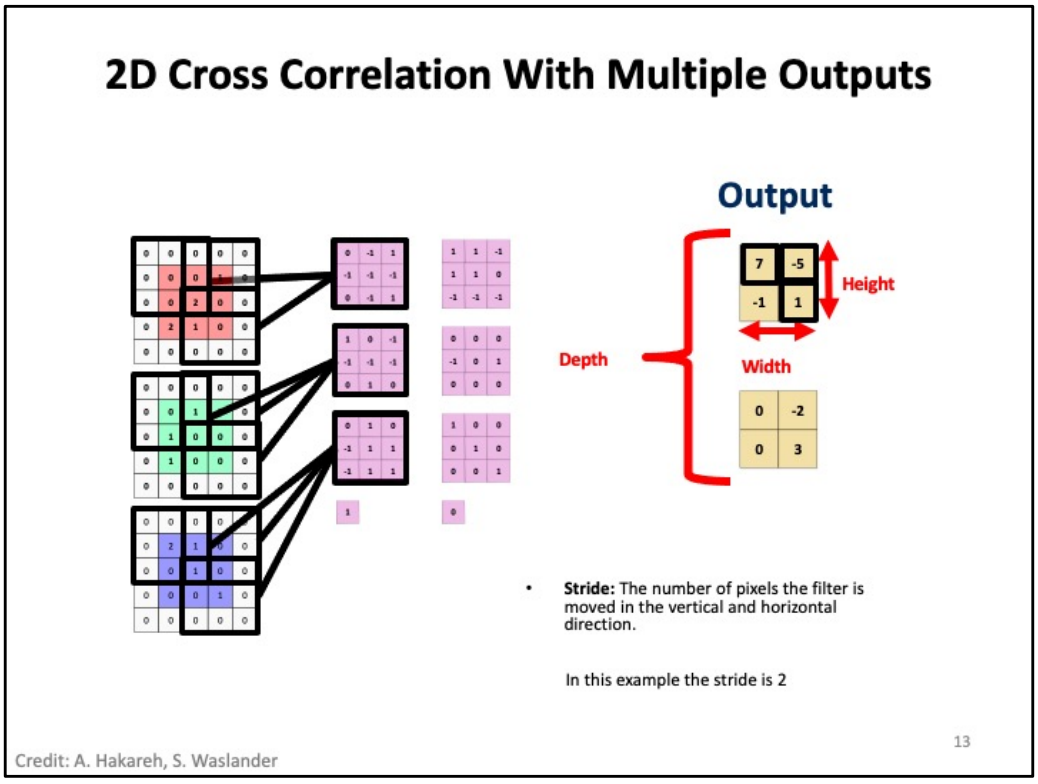


Since we not only want our convolutional layers to consume multi-channel inputs, but also to produce multi-channel outputs, we need as many multi-channel filters as there are channels in the output. For example, if we want the output to have two channels, we need two filters, one for each output channel.

The filter application to the input works exactly as in the image filtering case; we just apply each filter channel to the corresponding input channel, and then we add the responses across the channels to produce a single output.

Let's look at the diagram:

1. Let's assume the filter channels produce the following response for the top-left pixel of the input: 2, 0, 4, respectively. Remember that each response is just a weighted sum of the input pixels from the patch around the top-left pixel.
2. We then sum up the three channel responses and add the bias 1 to it, which gives us the output for filter 1, which is 7 (red circle, bottom). This is stored in the first channel of the output (small red circle, top-left), which corresponds to the first filter.
3. We repeat this process for all pixels and the first filter to compute the first output channel (large red circle, top-left).
4. Then we repeat this process for the second filter to compute the second output channel (blue circle).



As we slide the filters over the input, we have the choice to advance them in increments of one pixel, but we can also use larger increments.

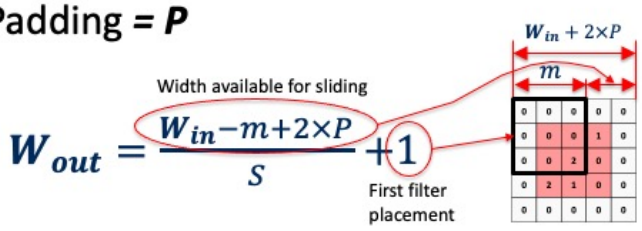
The increment is referred to as the stride. Larger strides shrink the input to a much smaller output size.

What is the stride we used in this example?

We have used the stride of 2, which cuts the input width and height in two.

Output Volume Shape

- Filters are size ***m* × *m*,** Number of filters = ***K* .**
- Stride = **S** ,Padding = ***P***



$$H_{out} = \frac{H_{in}-m+2\times P}{S} + 1$$

$$D_{out} = K$$

Credit: A. Hakareh, S. Waslander (extended)

14

It is helpful to have a formula that would allow us to compute the shape of the output volume, given the width and height of the input, the filter size, the padding size, the stride, and the number of filters.

The formula for computing the output width is as follows: $W_{out} = \frac{W_{in}-m+2\times P}{S} + 1$. Let's look what this formula means. We obtain the output width by dividing the input width available for sliding by the stride. This input width available for sliding equals the input width W_{in} enlarged by padding size on each side ($2\times P$), minus the filter width m (the width taken by the initial placement of the filter in the first column). We also need to add one to account for that first placement of the filter.

The height is computed in a similar way.

The output depth is equal to the number of filters.

Aside: Cross Correlation vs. Convolution

- A CNN does not care which one you use – it learns the right weights either way
  - Cross-correlating a filter with an input is equivalent to convolving the flipped (horizontally and vertically) filter with the input
  - See supplementary slides in the lecture on Image Filtering for the difference
- CNNs implement cross-correlation in practice

15

As I said before, a convolutional layer can be implemented using convolution or cross-correlation, and it does not matter which one we use, since the weights in the filters are learned.

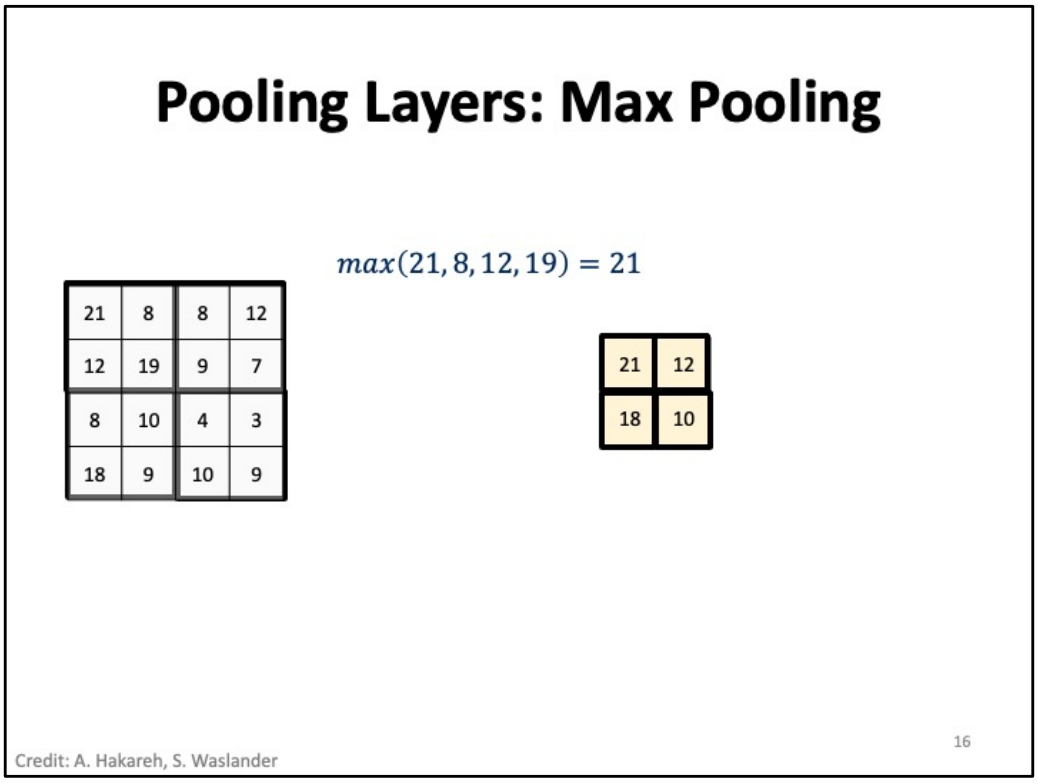
Recall from the image filtering lecture that cross-correlating a filter with an input is equivalent to convolving the flipped (horizontally and vertically) filter with the input. So if you replace convolution with cross-correlation, the network will discover the right filter, which will be the flipped version.
In practice, CNNs implement cross-correlation. There isn't really a difference; it's just by convention from image processing. Plus you can actually look at the filters in the first layers as images; they would be flipped if we used convolution.

Also,, it does not matter whether an implementation of a convolutional layer uses cross-correlation or convolution, but most libraries use cross-correlation.

Recall from the image filtering lecture that cross-correlating a filter with an input is equivalent to convolving the flipped (horizontally and vertically) filter with the input. Thus, cross-correlation can be expressed using convolution and vice versa: $h * f = h' \otimes f$ and $h \otimes f = h' * f$, where h' is the horizontally and vertically reflected (=reversed) version of h, e.g., if $h = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, then $h' = \begin{bmatrix} d & c \\ b & a \end{bmatrix}$. Since the filtered weights are learned, the right weight pattern is learned independently whether cross-

correlation or convolution is used. For example, if the network uses cross-correlation, it may learn to represent a face as h; and if it uses convolution, then it would learn h', i.e., the horizontally and vertically reflected version of the face.

As an aside, the reflection of a matrix can be expressed using the exchange matrix J, which is analogous to the identity matrix, but with 1s on the anti-diagonal. Left-multiplying by J swaps rows and right-multiplying by J swaps columns, e.g., $\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}a & b\\ c & d\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} = \begin{bmatrix}d & c\\ b & a\end{bmatrix}$.



The second type of layers in convnets are pooling layers, which aggregate information, and reduce the size of the input. Pooling layers can use different operations for aggregation, such as maximum or average, but we'll look at max pooling as an example.

The basic idea is to divide the input (on the left) into square patches, same as in filter application, and output the maximum pixel for each patch. The stride typically matches the filter size, as in this example, so that the filter applications do not overlap.
The result of applying the max pooling filter to the top-left patch is $max(21, 8, 12, 19) = 21$.
The resulting output is a smaller feature map or volume, and the stride determines the amount of shrinking.

Output Volume Shape

- Pool size ***m* × *m*.**
- Stride = **S.**

$$W_{out} = \frac{W_{in} - m}{S} + 1$$

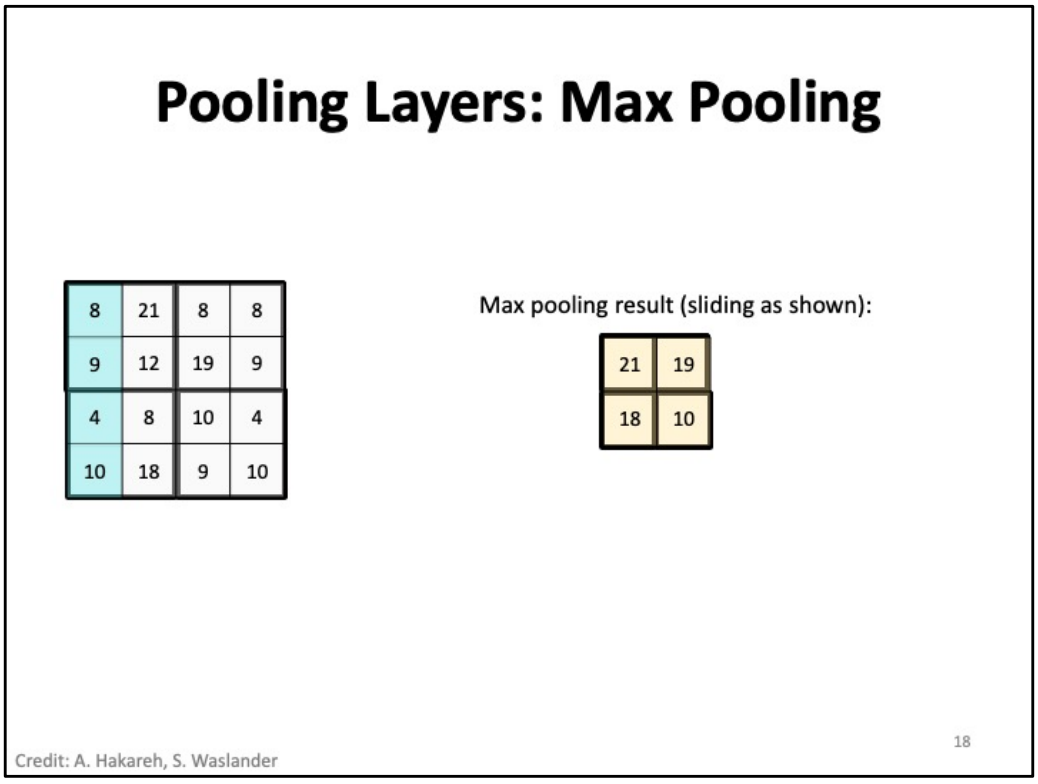
$$H_{out} = \frac{H_{in} - m}{S} + 1$$

$$D_{out} = D_{in}$$

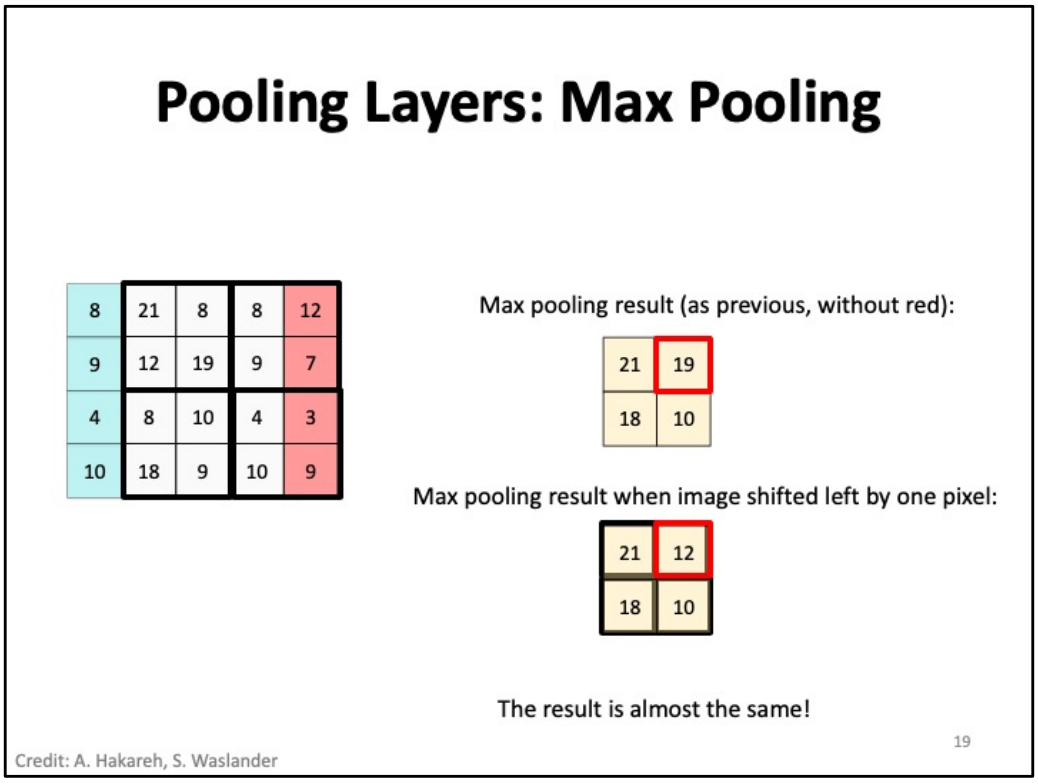
Credit: A. Hakareh, S. Waslander

17

The output shape is computed similarly as for convolutional filters, except that padding is not involved, since we do want to shrink the input (i.e., pooling is a form of downsampling).



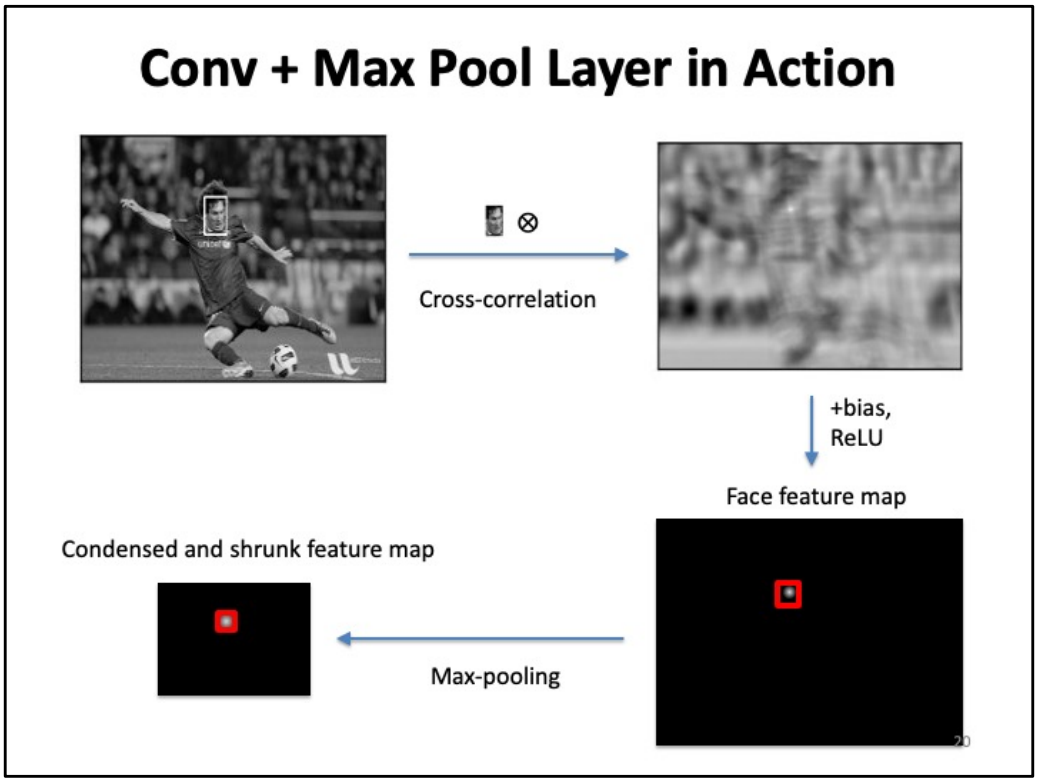
The key idea behind pooling is to aggregate or condense feature information, which will remove some detail. Pooling using maximum (as opposed to average) preserves the strongest signal.



A key property of pooling is that it makes outputs insensitive to small shifts of the input image. For example, when we shift a car in an image a bit, we still want a classification output for the car around the same location, i.e., we want the processing to be mostly invariant to small shifts.

Let's shift the input in the example on the right by one pixel to the left, so that we max-pool over the white and red pixels, but not the blue, which gives us the result shown bottom right (again, we select the maximum pixel in each patch).

Comparing both results, they are almost the same, with only one pixel value changed in the output!



Let's look at the conv and max pooling layers in action! For simplicity, the input and the output have each a single channel.

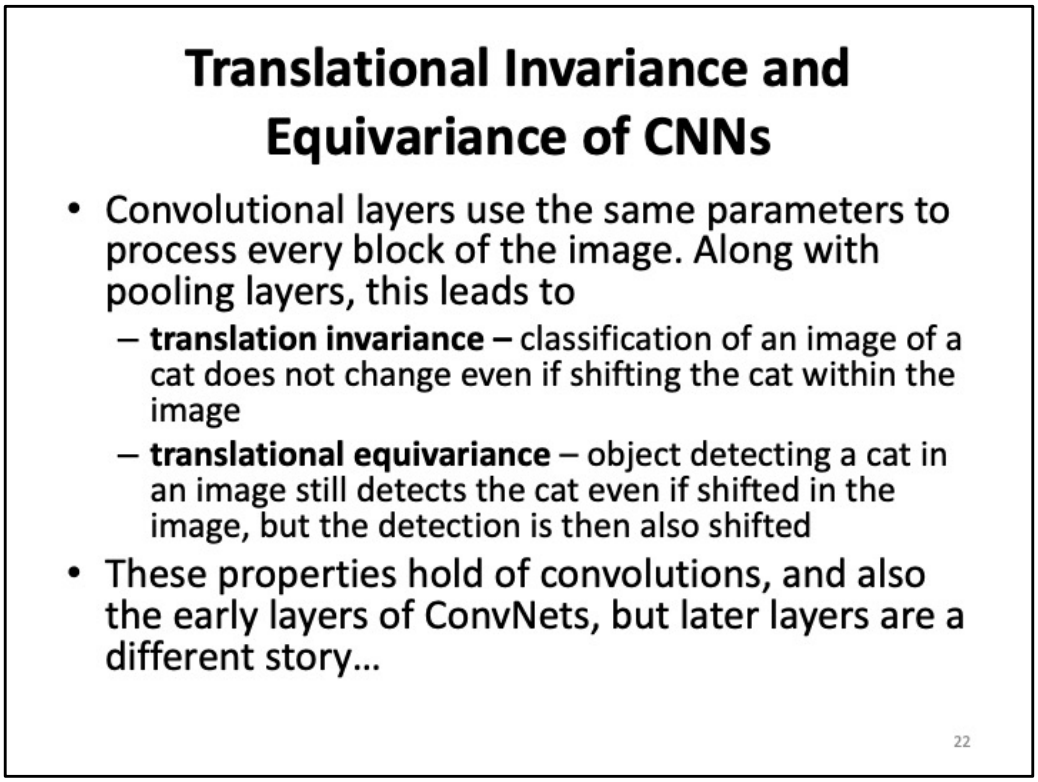
1. Given the grayscale input image (top left), a conv layer with the face filter would first cross-correlate the filter over the image (top arrow), producing a response (top right).
2. The conv layer would than add bias and apply a nonlinearity, e.g., ReLU (vertical arrow), which would give us a feature map with a clear activation for the detected face (bottom right).
3. Then the max pool layer (bottom arrow) would condense the input feature map into a smaller, summary feature map (bottom left), preserving the strongest responses. At this step, if we had multiple feature maps, say one for a different face (i.e., produced by a set of convolution filters), a different variant of pooling, so-called **cross-channel pooling**, could select the maximum signal not just spatially in 2D, but also across the feature maps to select the face that matches the strongest. Or we could use the so-called **global pooling** approach, which would compute one maximum or average feature per channel, to list all the face hypotheses found in the image.

Advantages of ConvNets

- Convolutional neural networks are by design, a natural choice to process images.
- Convolutional layers have **fewer parameters** than fully connected layers, reducing the chances of overfitting.
- The repeated application of filters over the 2D input captures **spatial relationships** within the image, and at different levels of abstraction throughout a deep network, leading to **hierarchical representations** of semantic concepts.

21

Partial credit: A. Hakareh, S. Waslander

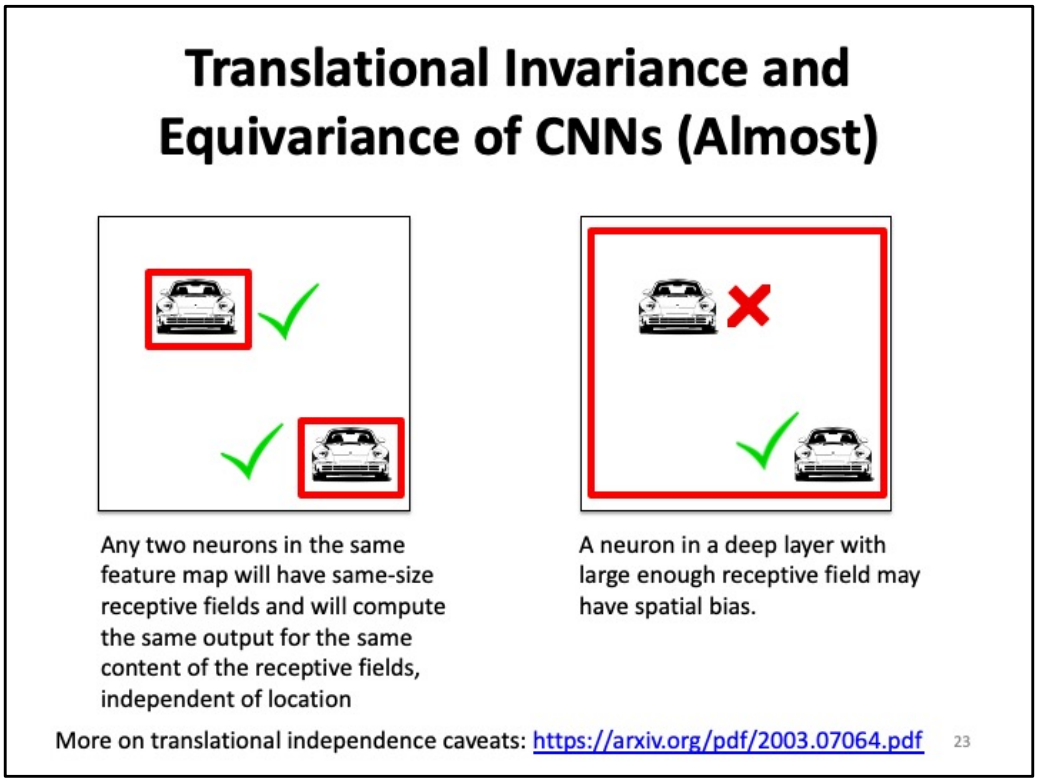


Another important property that CNNs can support is translational invariance and translational equivariance.

Translation invariance means that shifting the content of an image should not change the output of our network. We might want this to hold for a classifier. For example, if we have an image of a cat, and we shift the cat within the image, we want the image still to be classified as a cat.

Translational equivariance allows change to the output, but the change should correctly reflect the change to the input. For example, we want this property to hold for an object detector, where if we move the cat in the image, the cat should still be detected, but the bounding box should shift in the output to match how the cat was shifted in the input image.

These properties can be achieved with CNNs, but contrary to popular belief, they are not guaranteed. The properties clearly hold for convolutions (remember our discussion of shift invariance in the image filtering lecture?). They will also hold for early layers, but they may not hold for deep layers where neurons may have large receptive fields that may cover the entire input image. Let's look at this idea in more detail.



To understand how the receptive field impacts whether the network output is translationally invariant or equivariant, let's consider a simple example.

Assume that we have an image with two cars, and we analyze how these cars may be processed neurons in early layers (left) vs. deep layers (right).

Neurons in early layers will have small receptive fields, e.g., an output neuron from the first layer using a 3x3 filter would have a 3x3 receptive field. Any two neurons in the same early feature map will have same-size receptive fields that fit within the input image and will compute the same output for the same content of the receptive fields, independent of location (since each neuron will use same set of weights). In this example, if some content, like the car, is repeated in the input image, and if we had two neurons in some early feature map that would have the receptive fields as shown by the red boxes, the value of each neuron would be the same.

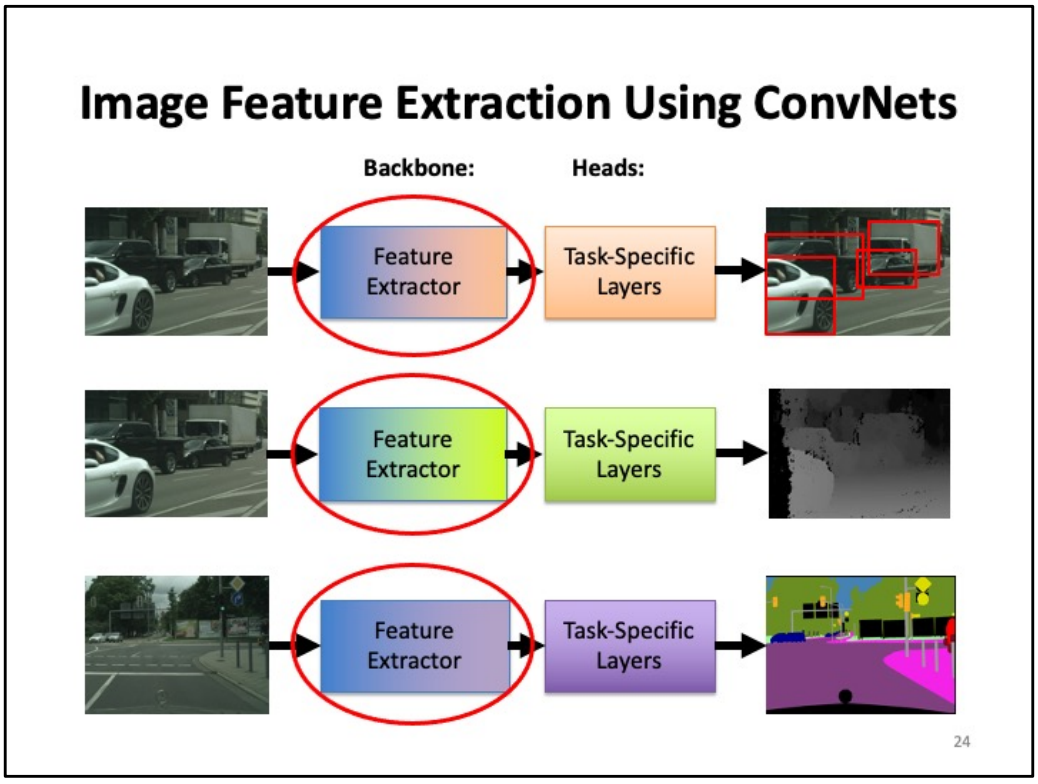
Now consider neurons in a deep layer, which likely will have very large receptive fields. The red box in the image on the right might represent the receptive field of one of these deep neurons (*). The neuron may now treat each location in the image differently, and for example, decide to detect cars at the bottom, but not at the top (perhaps the top was normally sky in the training dataset.). This is since potentially different weights are at play for the top vs. bottom of the input image. Other neurons in the deep layer may effectively look at the same parts of the image but look for

different cars.

Therefore, in general, CNNs may not be translationally invariant or equivariant, and we need to check for these properties in testing, e.g., by checking that cars in different locations are actually recognized. We also need to be careful with setting up the training to ensure that all locations where cars need to be detected are covered in the training dataset, potentially by using data augmentation.

You can read more on on the translational independence caveats for CNNs in the referenced paper.

(*) Note that the theoretical receptive fields for the deep neurons are often may times larger than the input image, but in practice the neurons also have much smaller empirical receptive fields that are induced by the specific weight values used in the computation of the neurons.



The main use of convnets in computer vision is to implement the so-called image feature extractors. An image feature extractor produces a set of feature maps, each representing a range of lower-level to higher-level features, which are needed for common computer vision tasks, like image classification, object recognition or semantic segmentation.

Let's illustrate the idea with some examples:

- The first part of an object detector (top) is a feature extractor, which is then followed by task-specific layers for object classification and localization.
- The same pattern applies for most image processing tasks, including depth prediction, where the task-specific layers perform pixel-wise depth regression (middle); and
- semantic segmentation, where the task-specific layers perform pixel-wise classification (bottom).

We sometimes refer to the feature extractor as the backbone, and the task specific layers as the head.

The feature extractor or the backbone can be shared over multiple tasks. Which leads to an architecture with a single backbone and multiple task-specific heads. Such an architecture is also referred to as a multi-tasking network. For example, the Tesla self-driving system uses such a multi-tasking architecture, which they refer to as hydra,

reflecting its many heads. The network has a single backbone, and heads for different tasks including object detection, road and lane marking detection, traffic sign detection, etc.

The Feature Extractor

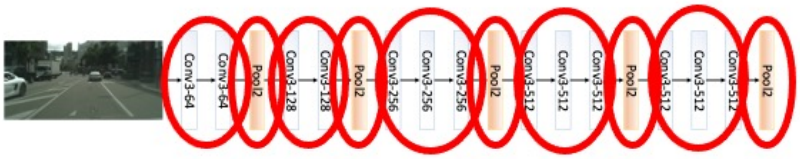
- Feature extractors are the most computationally expensive component in these pipelines (e.g., object detection).
- The output of feature extractors usually has much **lower width and height** than those of the input image, but much **greater depth**, to represent the many image features.
- Very active area of research, with new extractors proposed on regular basis.
- **Most common extractors are**: VGG, ResNet, DenseNet, and Inception.

25

Credit: A. Hakareh, S. Waslander

VGG Feature Extractor

- Alternating convolutional and pooling layers.
- All convolutional layers are of size 3x3xK, with stride 1 and 1 zero-padding.
- All pooling layers use the **max** function, and are of size 2x2, with stride 2 and no padding.



https://arxiv.org/abs/1409.1556

Credit: A. Hakareh, S. Waslander

26

VGG is one of the earliest and simplest architectures. It was proposed by Karen Simonyan and Andrew Zisserman of the Visual Geometry Group (VGG), University of Oxford, in 2014, and was a winning entry in the ImageNet benchmark at that time.
VGG has alternating convolutional and pooling layers, as shown in this figure. This figure shows the popular VGG-16 variant, which has 16 weight layers, 13 of which are convolutional layers (shown in this figure) and 3 more fully connected layers in the classification head (not shown).
All convolutional layers have filters of size 3x3xK, with stride 1 and 1 zero-padding, just as in our examples earlier. K is the depth. It is specified in each conv box, and it increases for deeper layers, as I will explain shortly. Each convolutional layer uses ReLU as activation function.
All pooling layers use the max function, and are of size 2x2, with stride 2 and no padding, again, just as in our example earlier.

VGG Feature Extractor

- All convolutions are of size 3x3xK, with stride 1 and 1 zero-padding. Convolutional output shape:
  - $\boldsymbol{W_{out}} = \frac{\boldsymbol{W_{in}} - m + 2 \times P}{S} + 1 = \frac{\boldsymbol{H_{in}} - 3 + 2 \times 1}{1} + 1 = \boldsymbol{W_{in}}$
  - $\boldsymbol{H_{out}} = \frac{\boldsymbol{H_{in}} - m + 2 \times P}{S} + 1 = \frac{\boldsymbol{H_{in}} - 3 + 2 \times 1}{1} + 1 = \boldsymbol{H_{in}}$
  - $\boldsymbol{D_{out}} = K$
- All pooling layers use the **max** function, and are of size 2x2, with stride 2 and no padding.
  - $\boldsymbol{W_{out}} = \frac{\boldsymbol{W_{in}} - m}{S} + 1 = \frac{\boldsymbol{W_{in}} - 2}{2} + 1 = \frac{\boldsymbol{W_{in}}}{\boldsymbol{2}}$
  - $\boldsymbol{H_{out}} = \frac{\boldsymbol{H_{in}} - m}{S} + 1 = \frac{\boldsymbol{H_{in}} - 2}{2} + 1 = \frac{\boldsymbol{H_{in}}}{\boldsymbol{2}}$
  - $\boldsymbol{D_{out}} = \boldsymbol{D_{in}}$

27

Credit: A. Hakareh, S. Waslander

We can apply our output size formulas to calculate the output shape for a given input shape, both for the convolutional and max pooling layers.

With the specific choice of 3x3 filters and size 1 zero-padding, each convolutional layer preserves the input width and height; however, the depth, which corresponds to the number of filters per layer, increases.

Each max pooling layer, with 2x2 size and stride 2, cuts the input width and height in half. The max pooling is applied per channel, and it preserves the depth.

The Feature Extractor

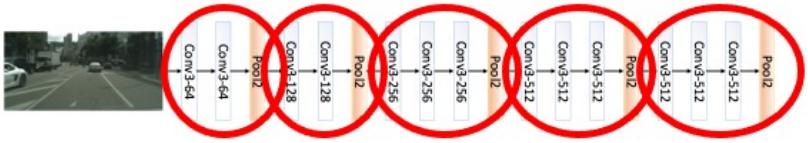


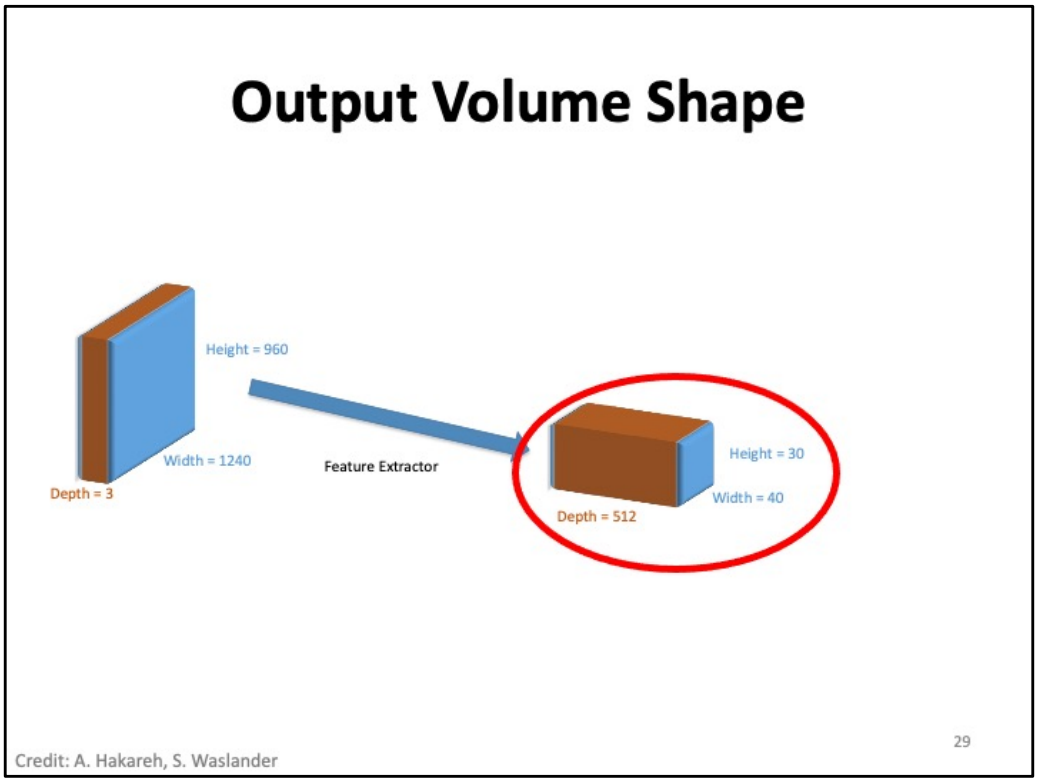
	Image	Conv1	Conv2	Conv3	Conv4	Conv5
Width	M	M/2	M/4	M/8	M/16	M/32
Height	N	N/2	N/4	N/8	N/16	N/32
Depth	3	64	128	256	512	512

28

Credit: A. Hakareh, S. Waslander

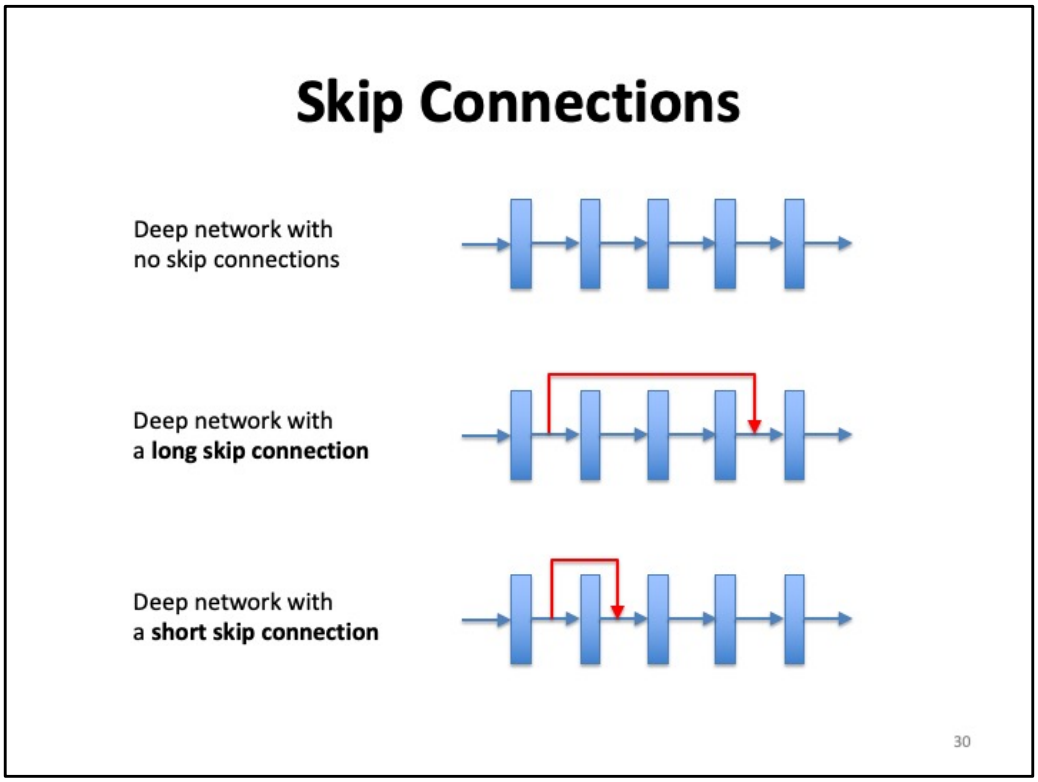
This table shows how the width and height is reduced by each block, which consists of two or three convolutional layers and one pooling layer. As already explained, each pooling layer cuts the width and height in half. As a result, the output feature maps have reduced resolution by 32 horizontally and vertically compared to the input image.

In contrast to the reduction of the resolution with depth, the convolutional layers in subsequent blocks have increasing number of filters, which go from 64 in the layers of the first block of the network, to 512 filters in the last two blocks. As a result, the number of channels in the feature maps increase with depth.



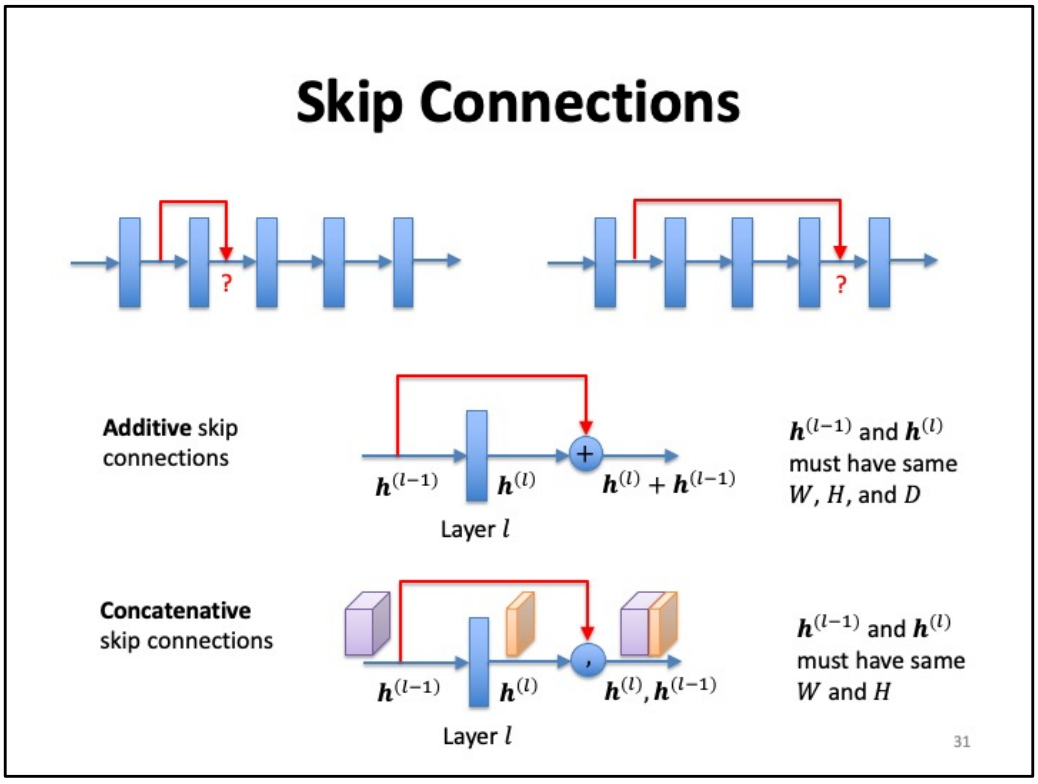
This figure visualizes how the VGG feature extractor summarizes the 1240x960 input image with 3 RGB channels in a stack of 512 40x30 feature maps.

When you train an image classifier that consists of a VGG feature extractor and a classification head with fully connected layers on a training set such as ImageNet, the extractor would learn a rich set of image features that are needed to perform image classification but can also be used other computer vision tasks.



VGG-16 was a huge advancement when it came out, but subsequent feature extractors performed even much better.

A key common feature in subsequent high-performance architectures are so-called skip connections. A skip connection is one that provides alternative routing in the network, by “skipping” one or more layers. A long skip connection skips many layers, and a short skip connection skips just one or two layers. Networks with skip connections often have many of them. Skips connections allow combining features of different scales and improve gradient flow through the network, which I will explain shortly.



One question is how a skip connection can be implemented.

We have two key variants.

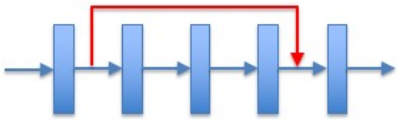
1. The first one is the so-called additive skip connection. In this type, the idea is to add to the output of a layer its input. The activations are added just elementwise. This works if the input and output have the same shape, that is, width, height, **and depth**.
2. The second variant is the so-called concatenative skip connection. Here we simply concatenate the input to the output, as shown in the figure. The concatenation happens by stacking along the channel dimension. As a result, the layer essentially augments its input with additional channels. Often, a layer may just add a few channels as a large stack of channels comes as an input. This concatenation works if the input and output have the same width and height. There is no constraint on depth in this variant.

The choice of one or the other mechanism is a hyperparameter, and it depends on the architecture.

Either variant can be used to implement short or long skip connections.

What do Skip Connections do?

- Improve gradient flow during training to avoid "vanishing gradient" problem
  - Improve convergence and speed up training
  - Allow training deeper networks
- Allow higher-level layers access a range of lower-level features
  - E.g., to consider details when classifying or detecting objects
- Bottom line: Enable higher performing models



32

This brings us to the question of why would we want to use skip connections.

The first reason is that skip connections improve the gradient flow during training and help us avoid the so-called "vanishing gradient" problem, which I will explain shortly. By addressing this problem, we improve convergence and speed up the training, and can also train deeper networks. Deeper networks may result in better features and predictions.

The second reason to use skip connections is to give higher-level layers access to a range of lower-level features. When looking at the stack of layers like in VGG, each layer has access to features produced just by its predecessor. This is a problem for many tasks, especially when higher-level layers need access to small details that are extracted by lower-level layers. Skip connections allow such access, as in this example (figure on the right) where low lever features flow directly to the top layer. Such details may improve classification, when small details are important to distinguish among otherwise similar classes; and they also help improve localization, such when recovering precise object boundaries in segmentation or regressing precise bounding boxes in object detection.

The bottom line of both uses is that they lead to better task performance.

Recall: Backpropagation

1. Forward pass
   - Compute all combined inputs $\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}$ activations $\boldsymbol{h}^{(l)} = \boldsymbol{g}(\boldsymbol{z}^{(l)})$ and the loss $L_{(i)}(\boldsymbol{y}_{(i)}, \boldsymbol{h}^{(d)})$ for datapoint i, $(\boldsymbol{x}_{(i)}, \boldsymbol{y}_{(i)})$
2. Backward pass
   a. Back-propagate gradients for $L_{(i)}$ wrt each activation:

$$\frac{\partial L_{(i)}}{\partial h_k^{(l)}} = \sum_{j=1}^{n_{l+1}} \frac{\partial z_j^{(l+1)}}{\partial h_k^{(l)}} \frac{\partial h_j^{(l+1)}}{\partial z_j^{(l+1)}} \frac{\partial L_{(i)}}{\partial h_j^{(l+1)}} = \sum_{j=1}^{n_{l+1}} w_{jk}^{(l+1)} g'\left(z_j^{(l+1)}\right) \frac{\partial L_{(i)}}{\partial h_j^{(l+1)}};$$

   b. And then wrt. each weight:

$$\frac{\partial L_{(i)}}{\partial w_{jk}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} \frac{\partial h_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial L_{(i)}}{\partial h_j^{(l)}} = h_k^{(l-1)} g'\left(z_j^{(l)}\right) \frac{\partial L_{(i)}}{\partial h_j^{(l)}}$$

33

Let's now look at the vanishing gradient problem and how skip connections help to address it.

In order to understand the problem, we need to briefly review the mechanics of backpropagation, which is used to compute loss gradient during training.

Recall that backpropagation starts with the forward pass (Step 1), which computes linearly combined inputs $\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}$ and activations $\boldsymbol{h}^{(l)} = \boldsymbol{g}(\boldsymbol{z}^{(l)})$ for each layer l and finally the loss $L_{(i)}(\boldsymbol{y}_{(i)}, \boldsymbol{h}^{(d)})$ for a datapoint i, $(\boldsymbol{x}_{(i)}, \boldsymbol{y}_{(i)})$.

Next is the backward pass (Step 2), which involves propagating gradients for the loss $L_{(i)}$ wrt. each activation and weight (we want gradient wrt. all weights $\boldsymbol{W}^{(l)}$ for all layers for training, but we need to flow the gradient across all layers via activations $h_k^{(l)}$ to do this).

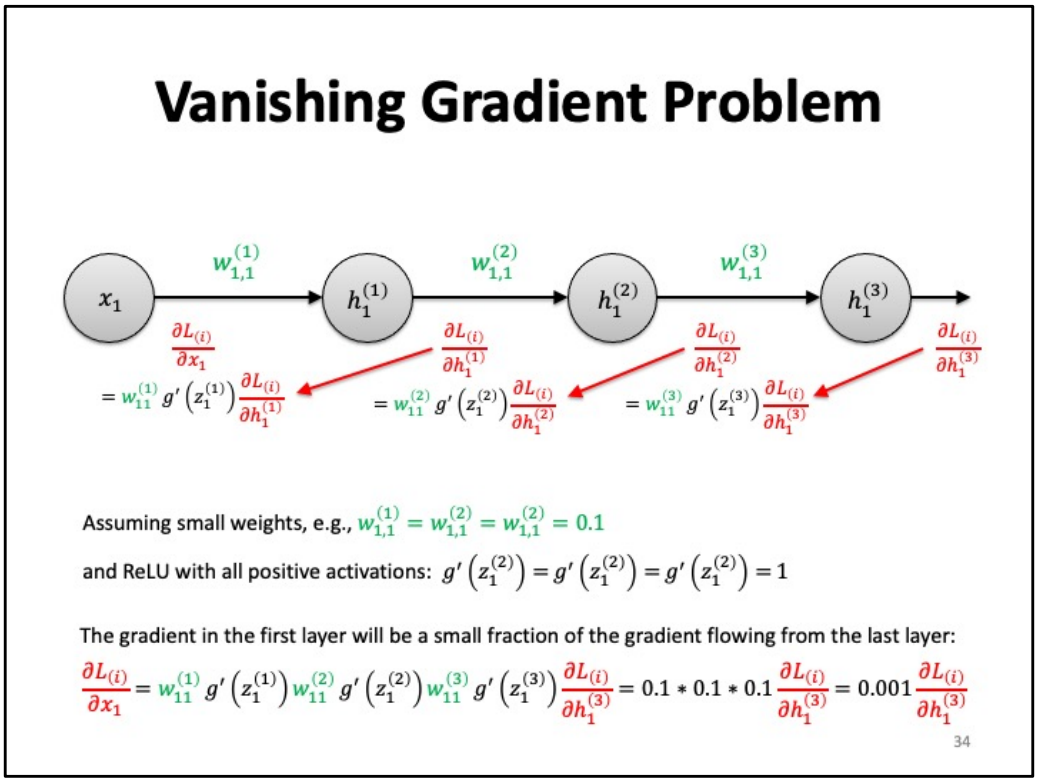
The basic template for propagating gradients for the loss $L_{(i)}$ wrt. each activation $h_k^{(l)}$, i.e., the activation k in layer l, is provided by the multi-variate chain rule (Step 2a), which adds up all the ways that $h_k^{(l)}$ may contribute to the loss via its connections to the units in the layer $l+1$. The addition of the contributions is

$\sum_{j=1}^{n_{l+1}} \frac{\partial z_j^{(l+1)}}{\partial h_k^{(l)}} \frac{\partial h_j^{(l+1)}}{\partial z_j^{(l+1)}} \frac{\partial L_{(i)}}{\partial h_j^{(l+1)}}$, where each contribution can be broken down as a multiplication of the derivatives of (i) the linear input combination $\frac{\partial z_j^{(l+1)}}{\partial h_k^{(l)}}$, which is $w_{jk}^{(l+1)}$; then the activation function $\frac{\partial h_j^{(l+1)}}{\partial z_j^{(l+1)}}$, which is $g'\left(z_j^{(l+1)}\right)$; and then the loss gradient $\frac{\partial L_{(i)}}{\partial h_j^{(l+1)}}$ wrt. to the next activation, $\partial h_j^{(l+1)}$. The last term $\frac{\partial L_{(i)}}{\partial h_j^{(l+1)}}$ is the gradient that flows from the downstream layers back into the layer $l+1$, and then gets converted into the gradient $\frac{\partial L_{(i)}}{\partial h_k^{(l)}}$ for the layer l using this equation. In summary, this conversion is achieved simply by multiplying the incoming upstream gradient $\frac{\partial L_{(i)}}{\partial h_j^{(l+1)}}$ with the current weight $w_{jk}^{(l+1)}$ and the derivative of the activation function $g'\left(z_j^{(l+1)}\right)$, which for ReLU is 1 when the activation is positive and 0 otherwise.

Next, to compute the loss gradients wrt. weights $\frac{\partial L_{(i)}}{\partial w_{jk}^{(l)}}$ (Step 2b), the gradients wrt. activations $\frac{\partial L_{(i)}}{\partial h_j^{(l)}}$ form the previous step are then multiplied by the activations $h_k^{(l-1)}$ at each weight $w_{jk}^{(l)}$ and the derivative of the activation function $g'\left(z_j^{(l)}\right)$.

Finally, the loss gradients wrt. weights is used to update the weights, but we will focus our analysis on the propagation of the gradients wrt. the activations. The key observation is that training speed for a given weight $w_{jk}^{(l)}$ depends on the magnitude of the loss gradient wrt. the weight $\frac{\partial L_{(i)}}{\partial w_{jk}^{(l)}}$, which is proportional to the the loss gradients wrt. activation $\frac{\partial L_{(i)}}{\partial h_j^{(l)}}$, which flows back into the connection with this weight $w_{jk}^{(l)}$.

If you are still confused by these calculations, please watch the two 3Blue1Brown videos I recommended in the previous lecture, which go over these equations in detail.



To explain the vanishing gradient problem, we'll limit our analysis to a sequence of 3 layers with just one neuron per layer. This will make the explanations simpler, but the reasoning generalizes to networks with multiple neurons per layer.

The first part of the back-propagation step flows the loss gradient for datapoint i taken wrt. the activations.

We start with the gradient wrt. to the activation in layer 3, $\frac{\partial L_{(i)}}{\partial h_1^{(3)}}$ (at the right end of the neuron chain).

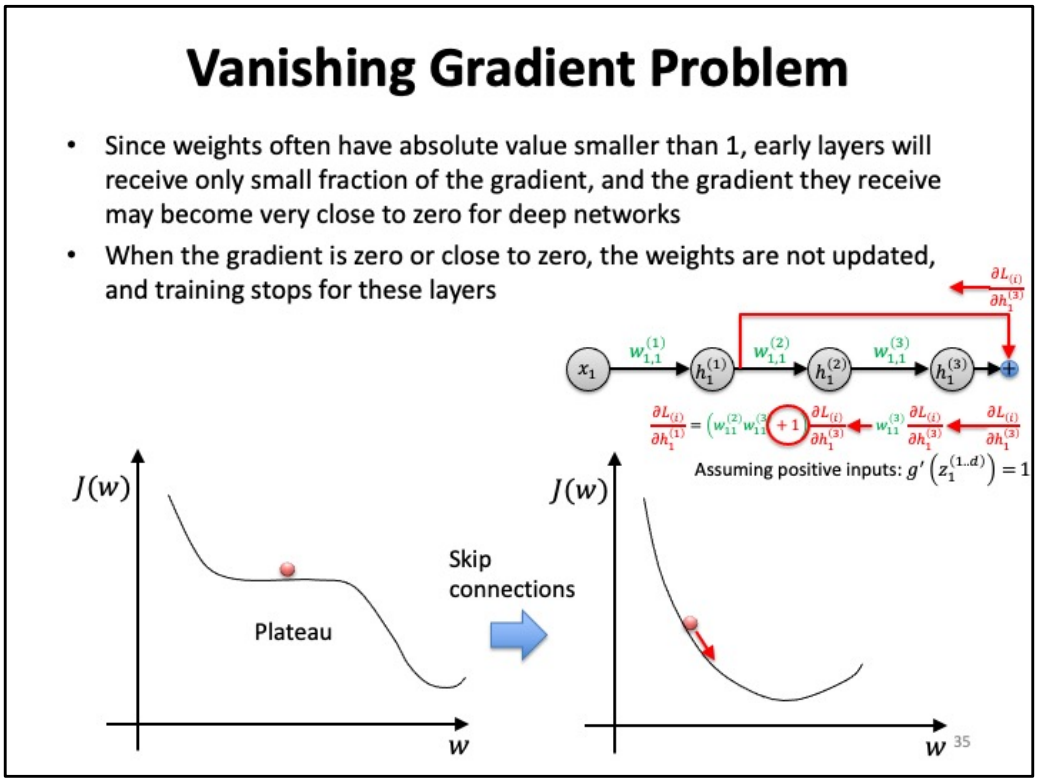
This gradient is then used to compute the gradient for the layer two activation $\frac{\partial L_{(i)}}{\partial h_1^{(2)}}$, by multiplying the incoming gradient by the weight connecting the layers and the derivative of the activation function, i.e., $\frac{\partial L_{(i)}}{\partial h_1^{(2)}} = w_{11}^{(3)} g'\left(z_1^{(3)}\right) \frac{\partial L_{(i)}}{\partial h_1^{(3)}}$.

This process is repeated until we have the gradient for each activation in the network.

For our example, let's assume that each weight has the value 0.1, i.e., $w_{1,1}^{(1)} = w_{1,1}^{(2)} = w_{1,1}^{(2)} = 0.1$. Note that weights often have values smaller than one. Let's also assume that each activation is positive, so the derivative of the ReLU activation function will be one.

With these assumptions, we can now compute the gradient in the first layer, which

effectively amounts to multiplying the gradient from the last layer by all the weights: $\frac{\partial L_{(i)}}{\partial x_1} = w_{11}^{(1)}\, g'\left(z_1^{(1)}\right) w_{11}^{(2)}\, g'\left(z_1^{(2)}\right) w_{11}^{(3)}\, g'\left(z_1^{(3)}\right) \frac{\partial L_{(i)}}{\partial h_1^{(3)}} = 0.1 * 0.1 * 0.1 \frac{\partial L_{(i)}}{\partial h_1^{(3)}} = 0.001 \frac{\partial L_{(i)}}{\partial h_1^{(3)}}$. As you can see, multiplying several small numbers gives us an even smaller number.



Now we are ready to define the vanishing gradient problem.

In essence, the loss gradient flows back to each weight during back-propagation, and the amount of gradient each weight receives governs its update or learning. Since early layers are likely to receive only small fraction of the gradient, they are prone to getting "stuck" during learning.

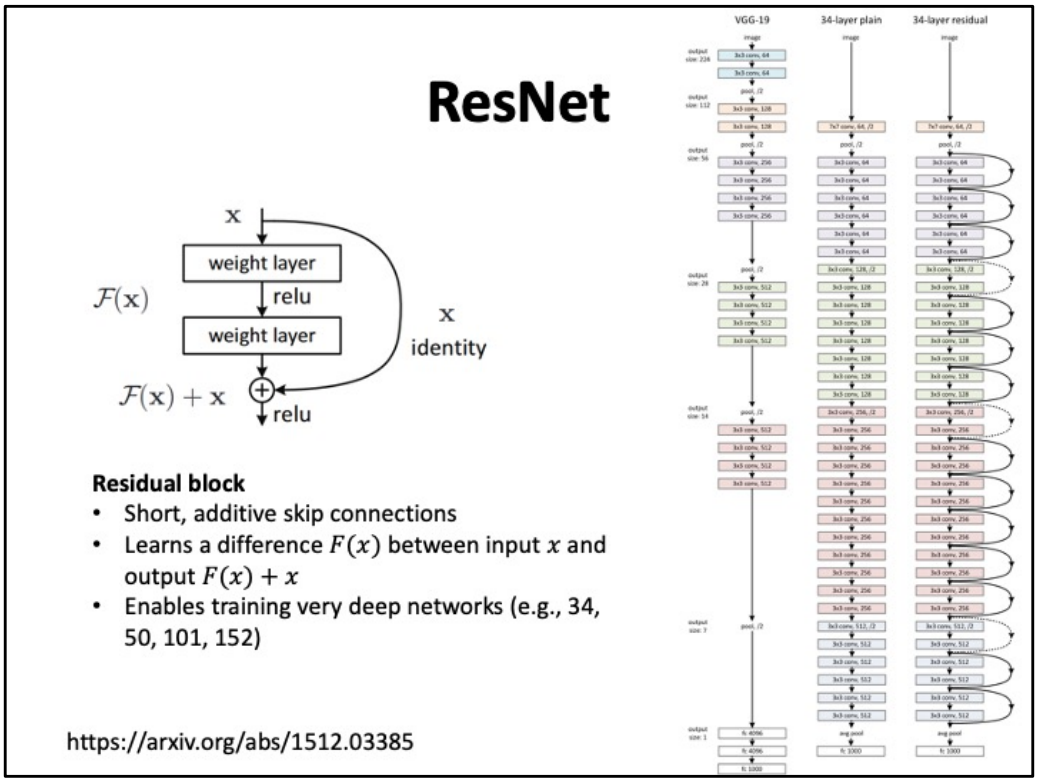
More precisely, in our example (previous slide, with no skip connection), you saw that the loss gradient wrt. activation $\partial h_1^{(1)}$ in the first layer, i.e., $\frac{\partial L_{(i)}}{\partial h_1^{(1)}}$, will be a small fraction of the gradient flowing from the last layer $\frac{\partial L_{(i)}}{\partial h_1^{(3)}}$, i.e., $\frac{\partial L_{(i)}}{\partial h_1^{(1)}} = w_{11}^{(2)} w_{11}^{(3)} \frac{\partial L_{(i)}}{\partial h_1^{(3)}} = 0.01 \frac{\partial L_{(i)}}{\partial h_1^{(3)}}$ (we assume positive inputs, i.e., $g'\left(z_1^{(1..d)}\right) = 1$). Since loss gradient wrt. the weight $w_{11}^{(1)}$ in the first layer, i.e., $\frac{\partial L_{(i)}}{\partial w_{11}^{(1)}}$, is proportional to loss gradient wrt. activation $\frac{\partial L_{(i)}}{\partial h_1^{(1)}}$ that flows back into the connection with that weight, i.e., $\frac{\partial L_{(i)}}{\partial w_{11}^{(1)}} = x_1\, g'\left(z_1^{(1)}\right) \frac{\partial L_{(i)}}{\partial h_1^{(1)}}$, the weight will receive little loss gradient, and will not be (significantly) updated during training. And for deep networks, some of these gradients in the early layers may, with rounding, become zero, i.e., learning stops for

these early layers.

We can visualize this is a plateau (see the plot on the left), where the absence of loss gradient wrt. a weight w implies that the gradient descent step magnitude is 0, so the the training is stuck. This example plot assumes that with a sufficiently large weight the training would recover, but if numerical rounding cuts off the faint gradient signal from earlier layers, the plateau might continue for indefinitely for any larger w.

Skip connections allow us to address this problem and make the loss landscape well shaped again (like in the plot on the right).
To see this, let's look at our sample sequence of neurons again, but now with a skip connection added (in red).
Assuming positive activations, the upstream gradient $\frac{\partial L_{(i)}}{\partial h_1^{(3)}}$ gets back-propagated by multiplying it with the weight between each layer, as previously explained. Now the key difference occurs for the loss gradient wrt. the first layer activation $\partial h_1^{(1)}$, i.e., $\frac{\partial L_{(i)}}{\partial h_1^{(1)}}$, because of the skip connection. This activation $h_1^{(1)}$ has two paths to influence the final loss: one path through layers two and three, and another direct path through the skip connection. In terms of gradient backpropagation, this means that the activation $h_1^{(1)}$ receives the upstream gradient not only via the weights of layers two and three, but also directly via the skip connection. This means that the gradient in the first layer is not just the final layer gradient multiplied by all the weights from layers two and three, but we now also have +1 due to the skip connection, i.e., $\frac{\partial L_{(i)}}{\partial h_1^{(1)}} = \left(w_{11}^{(2)} w_{11}^{(3)} + 1\right) \frac{\partial L_{(i)}}{\partial h_1^{(3)}}$. Thus, the addition of the skip connection which makes the overall gradient $\frac{\partial L_{(i)}}{\partial h_1^{(1)}}$ likely nonzero; in our example: $\frac{\partial L_{(i)}}{\partial h_1^{(1)}} = \left(w_{11}^{(2)} w_{11}^{(3)} + 1\right) \frac{\partial L_{(i)}}{\partial h_1^{(3)}} = 1.01 \frac{\partial L_{(i)}}{\partial h_1^{(3)}}$. Now the first layer can respond directly to the gradient signals coming from the last layer, not just the layer before it. This will also improve the loss landscape wrt. this early weight (like in the right plot).

NOTE: These notes were updated compared to the video to improve clarity. In particular, the updated example considers the update of the weight $w_{11}^{(1)}$ in the first layer and makes the involved gradients explicit.



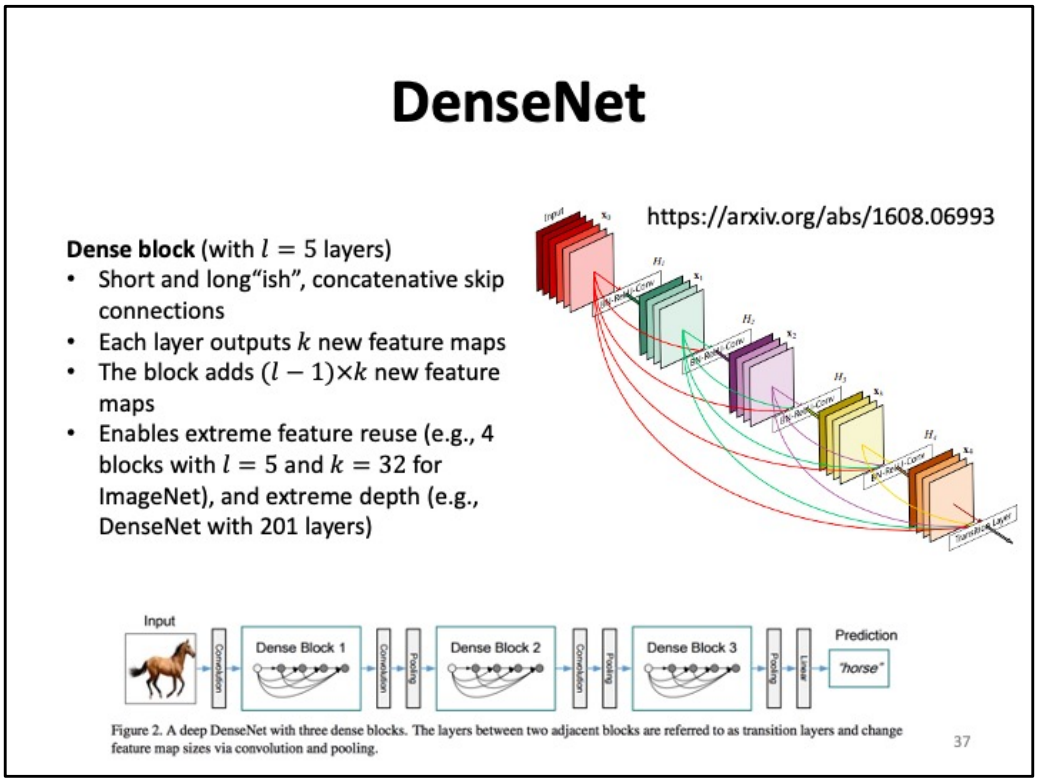
Let's now look at some sample architectures that use skip connections.

The first one is ResNet, which was proposed by a team from Microsoft Research Asia in 2015, one year after VGG, and was the top-performing architecture on ImageNet in that year.

ResNet consists of a sequence of so-called residual blocks. A residual block has two or three convolutional layers with a short additive skip connection (shown on on the left). The additive skip connection improves the gradient flow, as explained in the previous slide. In addition, it changes the task of the two layers. The two layers collectively implement a function F(x). When training, instead of learning to transform the input from scratch into some desired output, i.e., $z = F(x) = W_2\big(ReLU(W_1 x)\big)$, the layers with the skip connection learn the difference between the input and the output, i.e., $z = F(x) + x$ and thus $F(x) = z - x$, which the ResNet authors claim to be often an easier task.

Their paper reported on training very deep networks with up to 152 layers and outperforming the state-of-the-art architecture at that time. The architecture "34 residual" in the figure on the right shows the 34-layer ResNet variant from the original paper, which consists of 16 residual blocks. VGG-16 is shown on the left for comparison. Trying to train 34 layers without skip connections (middle) resulted in

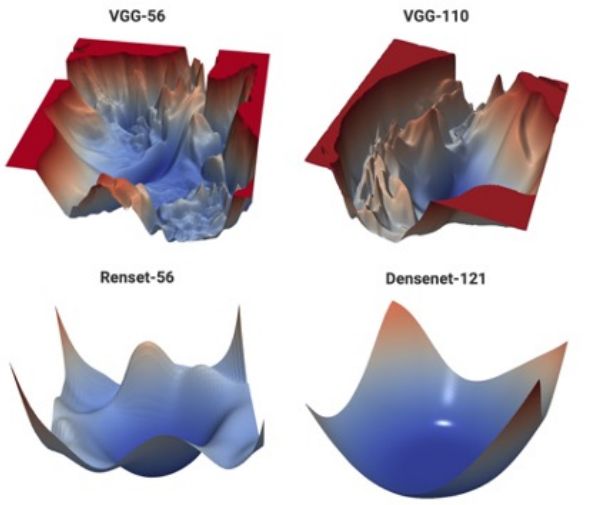
poorer performance than an 18 layer one; but with the skip connections added, they achieved top performance.



The second architecture using skip connections is DenseNet, which was proposed by a team from Facebook, in 2016, also achieving top performance on ImageNet in that year.

DenseNet consists of a sequence of so-called dense blocks. This figure shows a sample dense block with 5 layers, including the input.
The block uses both short and longer skip connections, which are, in contrast to ResNet, *concatenative*.
Each layer takes takes as input the stacked outputs from all the previous layers in the block and produces k new feature maps or channels.
Thus, each block adds $(l-1)\times k$ new feature maps (the minus 1 is for the input layer).
In addition to improving gradient flow, the skip connections in this architecture enable extreme feature reuse, where each layer gets to see the features from the previous layers in its block. The depth of the architectural variants evaluated in the original paper range between 121 and 264 layers, which are a lot of layers!

Impact of Skip Connections Loss Landscape

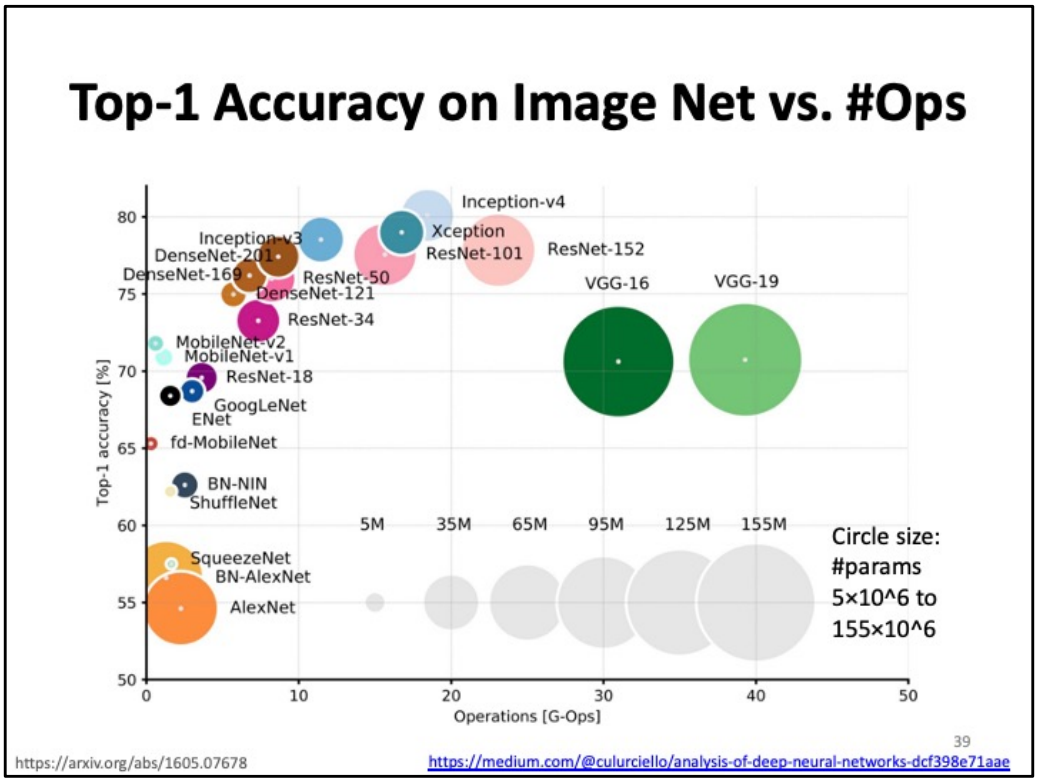


The top row depicts the loss function of a 56-layer and 110-layer net using the CIFAR-10 dataset, without residual connections. The bottom row depicts two skip connection architectures: Resnet-56 (identical to VGG-56, except with residual connections), and Densenet-121 (which has a very elaborate set of skip connections). Skip connections cause a dramatic "convexification" of the loss landscape.

38

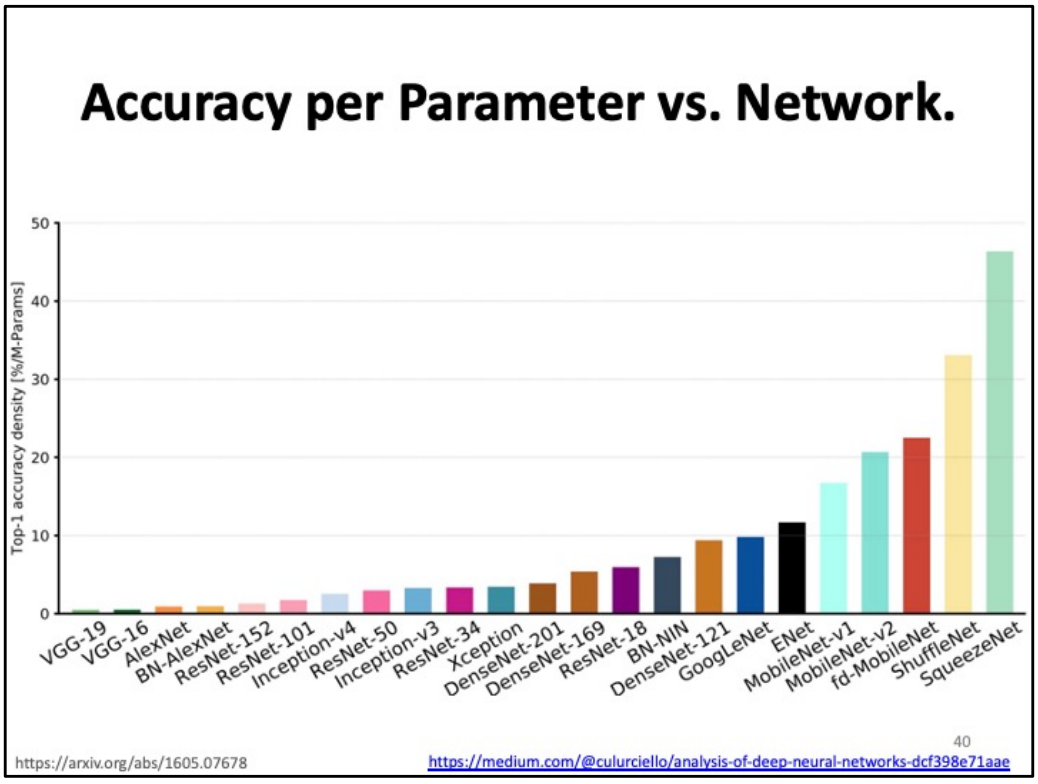
https://www.cs.umd.edu/~tomg/projects/landscapes/

Here is again the visualization of the loss landscape for the discussed architectures, which I already showed in the previous lecture.
The top row shows the loss landscape for VGG, and the bottom row shows sample landscapes for ResNet and DenseNet.
You can clearly see that the architectures with the skip connections have much smoother and thus better loss landscapes.



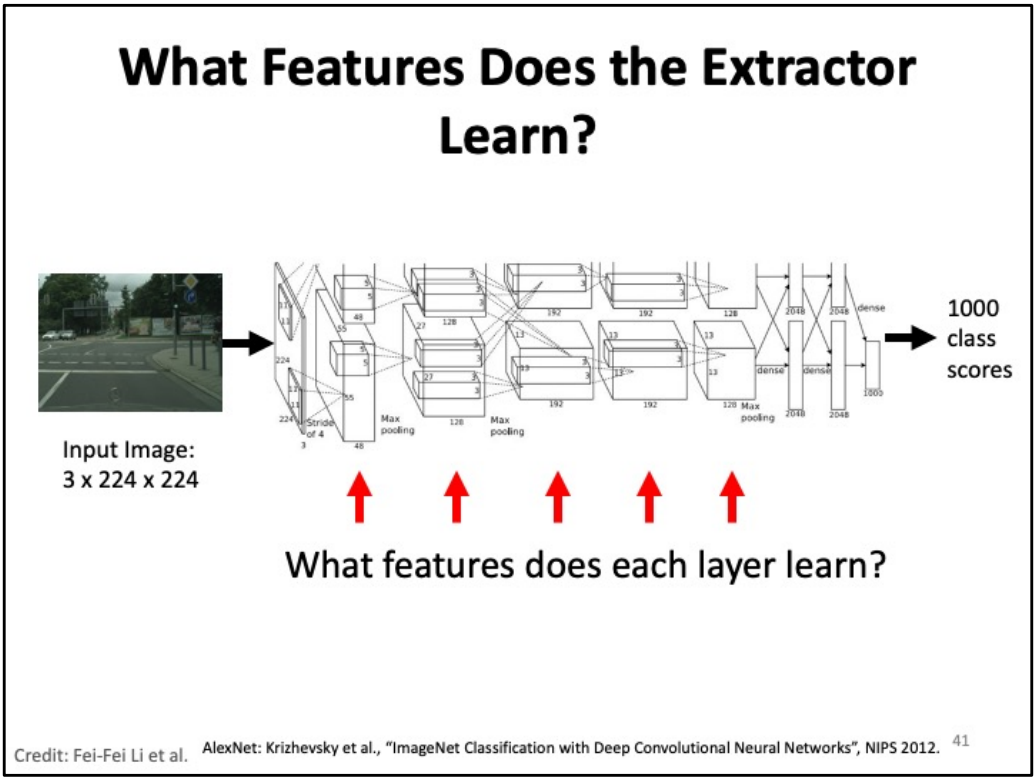
This chart compares the performance of different network architectures on ImageNet. The horizontal axis represents the computational cost in giga-operations per inference (i.e., forward pass). The vertical axis represents the classification accuracy on ImageNet. This is the so-called top-1 accuracy, meaning that only the class with the top score is counted and checked whether it is the correct lable. The size of the blobs is proportional to the number of network parameters, which impacts memory consumption. The mapping of the size to number of parameters is shown in the legend in the bottom, spanning from 5×10^6 to 155×10^6 params.

The best performing networks are in the upper left corner. They include Densenet, ResNet, and Google's Inception.
VGG is clearly very suboptimal.
MobileNet and SqueezeNet are interesting, since they are very lightweight and still achieve good performance, as we'll see next.



This chart compares the different architectures in terms of accuracy per parameter. This is an efficiency metric that highlights the capacity of a specific architecture to better utilise its parametric space.

1. Models like VGG and AlexNet are clearly oversized, and do not take full advantage of their potential learning ability.
2. On the far right, DenseNet-121 and ResNet-18, do a better job at “squeezing” all their neurons to learn the given task.
3. But the winners in this category are networks like SqueezeNet, ShuffleNet and MobileNet. These architectures have their own tricks to optimize for size. For example, MobileNet uses so called separable convolutions, which use 1x1 bottleneck convolutions. SqueezeNet also uses 1x1 rather than 3x3 convolutions and it also uses model compression techniques.

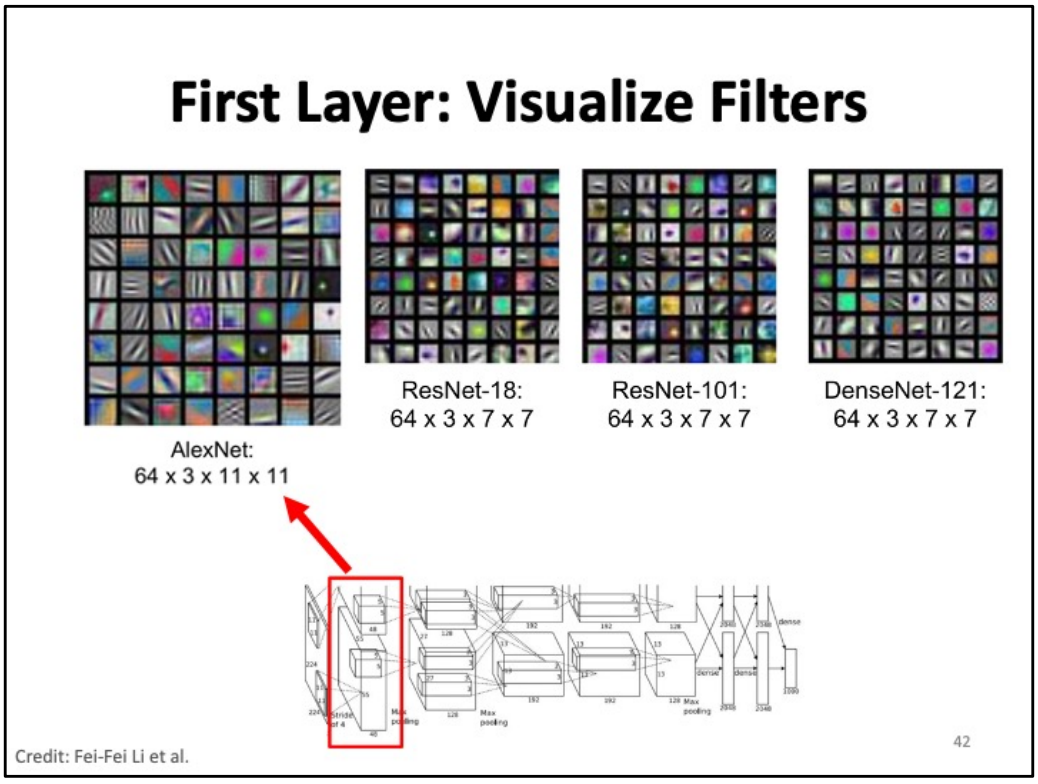


In the remainder of this video, we'll peak under the hood of a trained CNN to get a sense of what types of features it learned in each layer.

We'll use AlexNet as an example. AlexNet was designed by Alex Krizhevsky, in collaboration with Ilya Sutskever (who is currently the chief scientist at OpenAI) and Geoffrey Hinton (one of the fathers of deep learning). The network won the ImageNet in 2012 by a large margin (10% error reduction) and was one of the first deep CNNs implemented on a GPU.

AlexNet contains a feature extractor with five convolutional layers with ReLUs and three max pooling layers, and a classification head with three fully connected layers. The diagram shows the dimensions of the feature-map volumes (the big blocks) and the filters (the skinny ones). This diagram of the AlexNet architecture from the original paper also shows two parallel branches; the network was split into these two halves, in order to fit them into two GPUs.

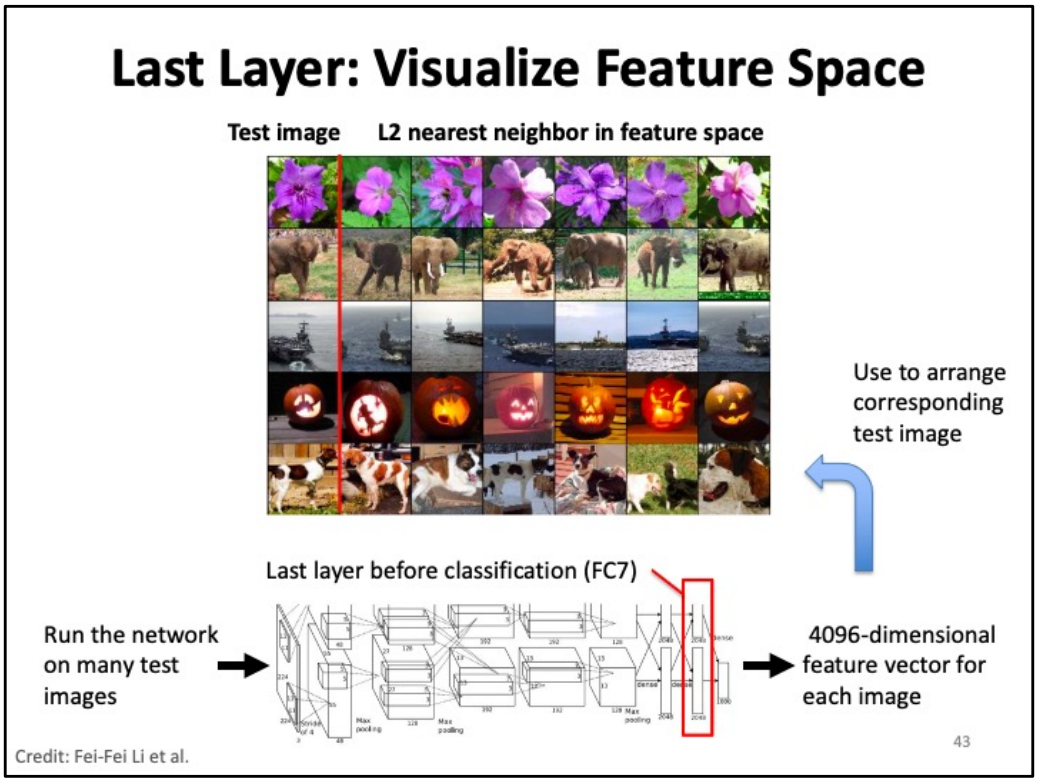
In the considered example, the network is trained on ImageNet (2012), which a large, publicly available dataset of images, with each image classified into one of 1000 classes. The training split has 1.2 million images, the validation set has 50 thousand and the test split has 100 thousand images.



The features learned by the early layers are easiest to inspect. We can simply view the filters as images. Recall that the filters are pattern matchers, and the early layer filters operate on the input image, so they directly represent the shapes they look for (particularly when we use cross-correlation, so the patterns are not flipped).

AlexNet has 64 11x11 filters (shown on the left), each with 3 channels. We can see that the filters look for primitive features, like edges of different orientations, corners, and textures.

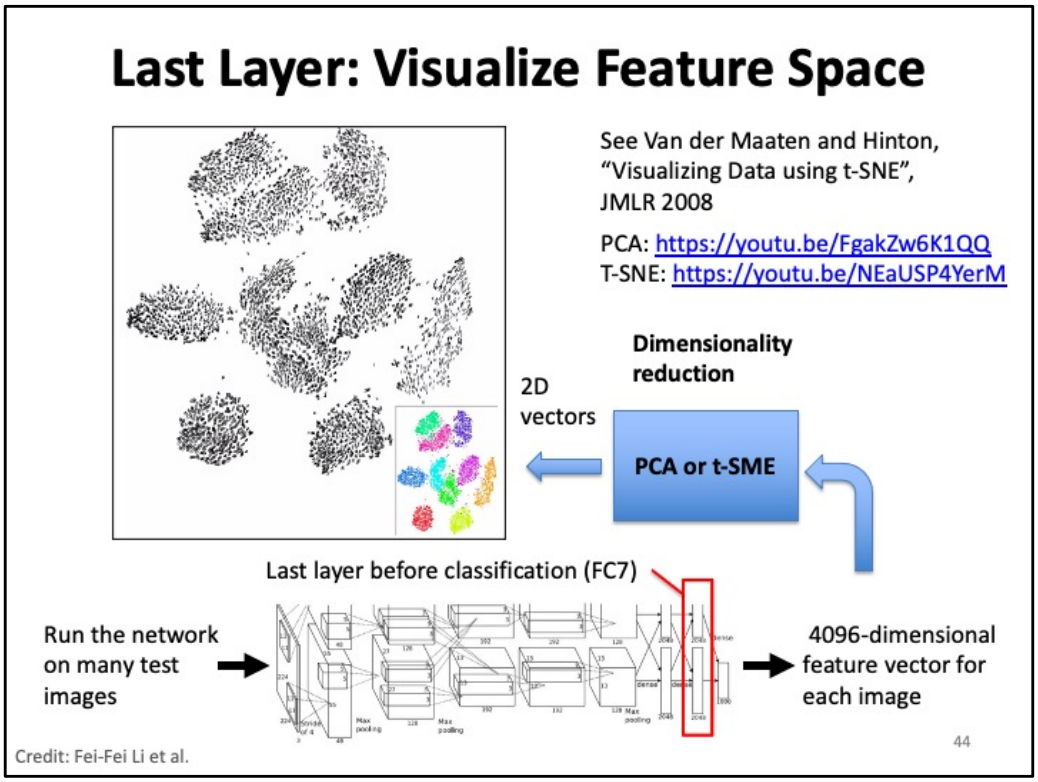
Interestingly, if we look at the first-layer filters of the other architectures trained on ImageNet, such as ResNet and DenseNet, these networks discover very similar basic low-level patterns. Note that their filters are 7x7 rather than 11x11. These filters will vary even over multiple training sessions of the same architecture, but the kinds of patterns that emerge in the early layer are similar.



Peaking into the later layers and the high-level features they compute is challenging, since we cannot simply look at the filters and make sense of them. This is because they are expressed in terms of the features from the earlier layers.

Therefore, researchers have developed several methods to help us with this task.

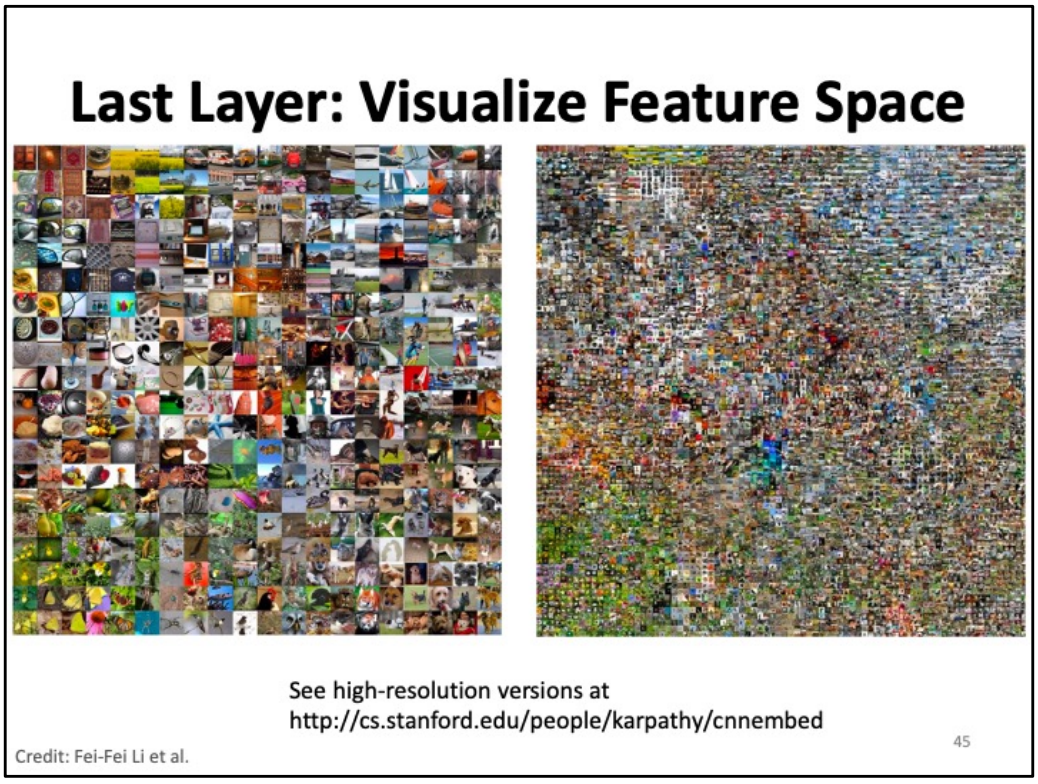
As a first method, let's use nearest neighbor clustering to investigate the penultimate layer, that is, the last layer before the classification output layer. This layer produces a set of very high-level features, which are sufficient for the linear classification of the output layer to separate the images according to their classes. When we input an image into the trained network, the penultimate layer produces a 4096-dimensional feature vector, that is, a vector with 4096 numbers, encoding the high-level image features. We cannot make any sense of these numbers by just looking at them, but we can look at the images that have similar feature vectors.
Thus, for each test image (left column), we can look for other images with a similar feature vector, where similarity is measured using the L2 distance norm, that is, the square root of the sum of squared differences of the features. We see that visually similar images have also similar feature vectors.



Another idea is to use more sophisticated clustering algorithm to show how the images may be distributed in the feature space.
Since the we cannot directly visualize a space with 4096 dimensions, we need to use some dimensionality reduction method, such as principle component analysis (PCA) or the t-SNE method, to identify two or three dominant direction in the 4096-dimensional space that separates the images best.

This visualization shows how the features computed by the network organize the input images into clearly separable clusters in two dominant dimensions, where different clusters represent very distinct classes (indicated by color).

BTW, PCA is a standard unsupervised machine learning method. If you are unfamiliar with it, I recommend watching this video. I've also linked a video on t-SNE. Finally, the most recent and popular method is Uniform Manifold Approximation and Projection (UMAP).

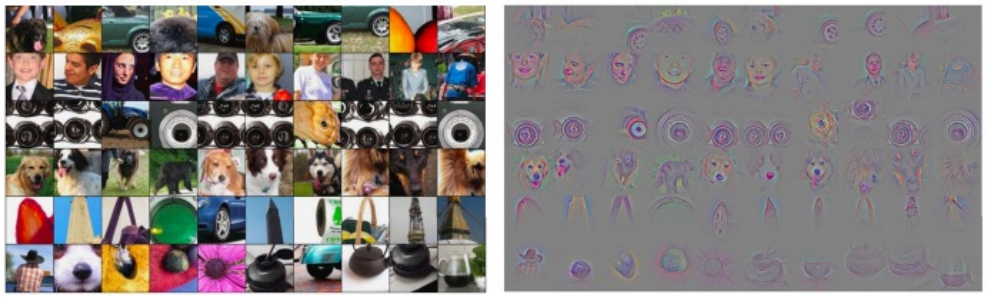


This is a visualization of a large number of images from ImageNet placed according to their feature vector similarity. This visualization used t-SNE to identify a two-dimensional mapping of the 4096-dimensional feature vector space that preserves the L2 distance among the image vectors from the higher-dimensional space.

We can see that images from similar categories cluster together, such as dogs in the lower right corner, or boats in the upper right corner. But there is also some mixing, like the two polar bears among the dogs, or the airplanes among the sailboats.

Intermediate Layers: Activation Analysis

- **Gradient-based explanation**:
  - Determine pixels of **specific** input image that activate a given neuron in some layer
  - This is determined using the gradient of the neuron activation wrt. each input pixel
  - We can highlight each pixel in the input according to this gradient
  - The pixels with the highest gradient are those the neuron pays most attention to



Each row corresponds to a selected neuron in an intermediate layer.

Each cell on the right shows the pixels from the original input (on the left) that maximally activate the neuron, computed using Guided Prop

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014

46

Now let's look at the features learned by the intermediate layers. Here again we need some special explanation techniques since we cannot make sense of these intermediate features by looking at the filters or the feature maps directly.

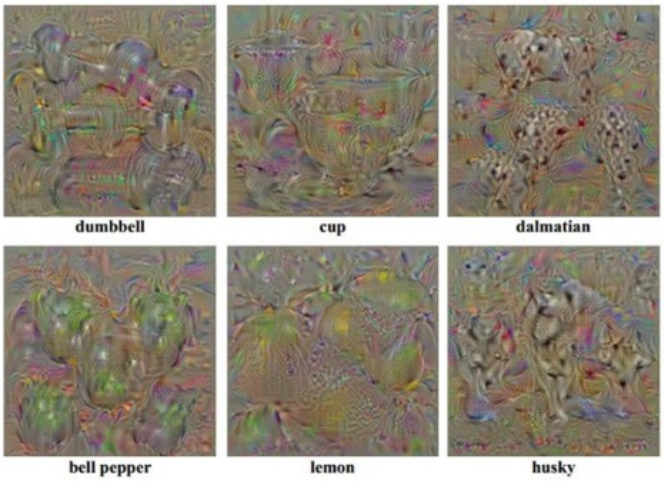
A common method is to use the gradient of the activation of a neuron that we want to explain, with the gradient taken wrt. each input pixel.
This approach allows us to identify which input pixels a given neuron is sensitive or pays attention to. The key point to realize is that this method explains what the given neuron pays attention to for only a specific, concrete input image. It does not necessarily tell us what the neuron would pay attention to in another image. We call this type of explanation local, because it is specific to an input. The activation gradient of the neuron wrt. to a pixel can easily be computed using backpropagation, and then the gradient value for each input pixel can be shown as an intensity image.

Let's look at an example. This illustration shows input images on the left and the corresponding gradient explanations on right. Each row corresponds to a selected neuron in an intermediate layer (for example, a neuron in some channel of an output of an intermediate layer; note that each filter producing a given channel may be specializing to look for a specific type of feature or object). Each cell on the right shows the pixels from the original input (on the left) that maximally activate the neuron. For example, the top row shows a neuron that clearly responds to car wheels. The second row shows a neuron that responds to faces, etc. Thus, we see

that neurons in higher level layers specialize to activate for higher level features. The specific technique that was used to create these visualization is called GuidedProp. It follows the familiar gradient back-propagation principle to compute the gradient of activations wrt, input pixels, but has specific enhancements to deal with max pooling layers. Max pooling layers perform down sampling, i.e., they lose detail, but we still want to have crisp representations in the explanation image. The technique saves information for max-pooling layers in order to be able to recover the lost details. You can read about this technique in the cited paper.

Intermediate Layers: Activation Analysis

Activating pixels for entire classes via gradient ascent



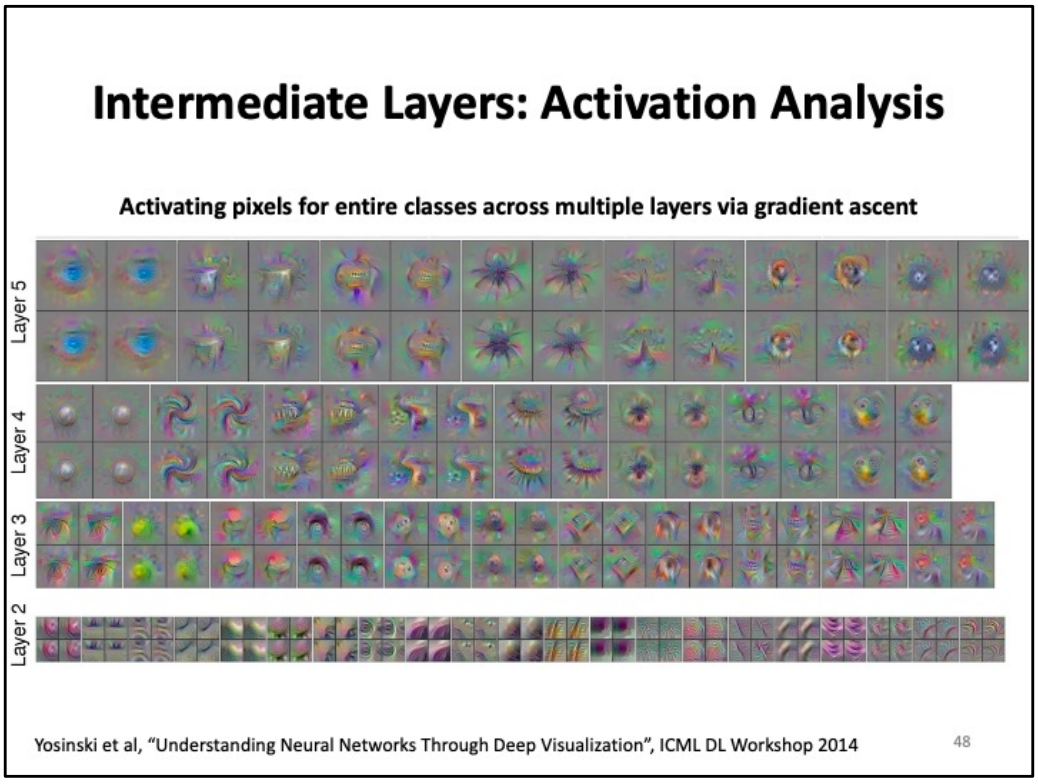
Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

47

While the previous technique showed local explanations, that is, parts of a concrete image that a neuron pays attention to, we might be also interested in so-called global explanations, which would tell us all the different types of inputs that a given neuron would pay attention to.

One such technique is gradient ascent. It is a generative technique. More precisely, it generates images that visualize patterns that would maximally activate a given neuron. This is done simply by starting with all zero image and then performing gradient ascent for each neuron of interest, such as an output neuron for a class of interest, with respect to the input pixels. This is much like gradient descent in training, except that we now adjust input pixels rather than weights and we want to maximize the network output for a given class or the activation of a given neuron, rather than minimizing the loss.

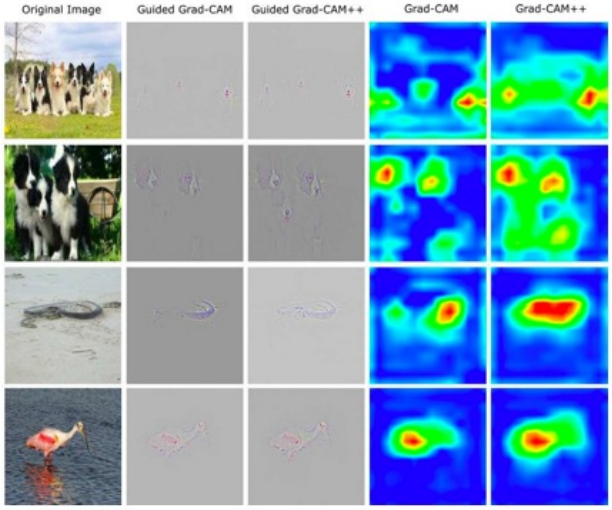
The resulting images are somewhat hard to interpret, but they are a kind of a jumble of all the patterns that are given neuron looks for. In the shown visualizations, we are explaining the activation of output neurons. For example, the first figure shows the patterns that would activate maximally the neuron that corresponds to the class dumbbell. Similarly, we have maximum activation images for the classes cup, dalmatian, bell pepper, lemon, and husky.



The gradient ascent method can also be applied to explain neurons in intermediate layers.

This visualization shows generated images that maximally activate specific neurons in different intermediate layers of a network trained on ImageNet. We see that neurons at higher levels focus on increasingly complex patterns of concepts. While layer 2 is still focused on small features and textures, layer 5 includes large concepts, such as a spider.

Which Part of an Image Contribute to the Network's Decision for a Given Input?



A. Chattopadhay, A. Sarkar, P. Howlader, V.N. Balasubramanian. Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. WACV, 2018

49

The explanation techniques have also some practical value beside giving us some impression of what CNNs learn. These techniques are being developed in the field of so-called explainable AI.

For example, the gradient-based technique to identify parts of the input that a given neuron responds maximally to using gradient forms a basis for several concrete explanation methods, such as several variants of the so-called Grad-CAM. These methods output so-called saliency or attention maps, which highlight the potion of the input that the network uses to make its decision. In this example, we see how these methods highlight parts of the input image that were responsible for the given classification decision, such as in the case of the dogs. Grad-CAM++ is a state-of-the-art technique. In general, we want such techniques to be faithful, that is, they should provide correct explanations.

These techniques help diagnose potential problems or biases in object classifiers or detectors. For example, if we discover that a network pays attention to the wrong part of the input image to make its decision, even if the decision is correct, we need to address this problem to make it generalize better. This insight may tell us what training images we may be missing.

Key Takeaway Points

- ConvNets were the **first** neural network models to perform well on computer vision tasks at a time where other feedforward architectures failed.
- ConvNets were the **first** neural network models to solve important commercial applications, such as handwritten digit recognition in the early 1990s [LeCun et al.]

50

Credit: A. Hakareh, S. Waslander

Key Takeaway Points

- ConvNets replace the multiplication with the full-connectivity weight matrix with a convolution or cross-correlation with a small filter that applies the same weights when sliding across the input.
  - This results in far fewer weights, makes the representation spatially aware wrt. the input grid, and affords translational invariance.
- ConvNets consist of convolutional layers, with multiple multi-channel filters, and max-pooling layers, which summarize and down-sample the input.
- ConvNets are used to implement the feature extractor, which is the workhorse in many CV tasks; each task requires a task-specific head.
- Several ConvNets architectures exist, including VGG, ResNet, and DenseNet; and state-of-the-art ones use skip connections to improve gradient flow and communicate low-level features to higher-level layers.
- Early layers in ConvNets focus on low-level features, and subsequent layers focus on increasingly high-level features; and local and global explanation techniques can give us such insights.

51