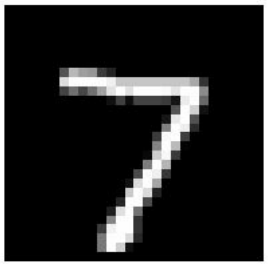
Deep Network Development

Lecture #8

Viktor Varga
Department of Artificial Intelligence, ELTE IK

Last week - MLP for handwriting recognition



x

Tuning the parameters

J (categorical crossentropy)

n = 784

h

k = 10

x

ŷ

softmax

y
0
0
0
0
0
0
0
1
0
0

y^
0.01
0.11
0.04
0.01
0.02
0.01
0.01
0.78
0.0
0.01

argmax → 7

Last week - MLP for handwriting recognition

Previously:

E.g., logistic regression for estimating cholesterol levels

→ 3 input variables, 4 parameters, several hundred data points

<

Now:

MLP with a single hidden layer for the classification of handwritten digits

→ 784 input variables, 636k parameters, 60k data points

>

What can happen when we have too many parameters?

→ **Overfitting**

Last week - How to deal with overfitting?

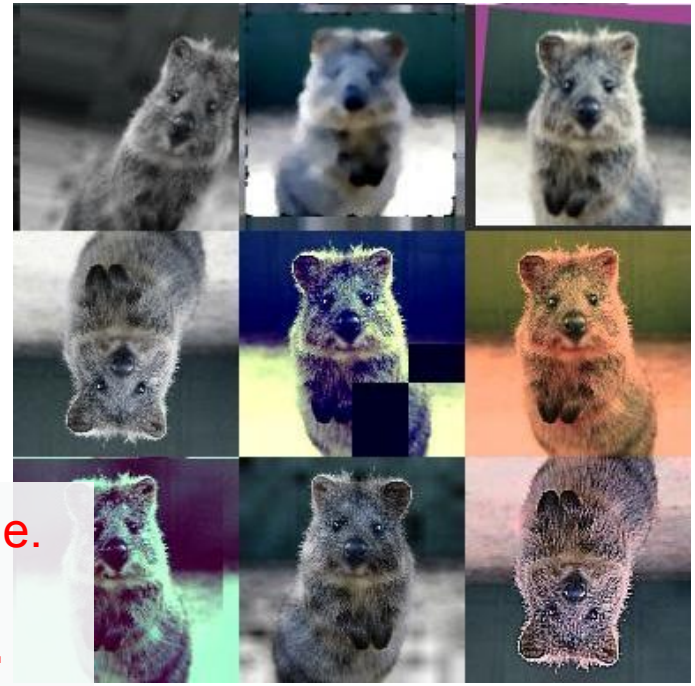
How can we avoid overfitting?

- Use a **simpler model** (e.g., fewer parameters)!
- Regularization methods (e.g., L2 regularization)
- Obtain **more training data**!
- **Early stopping**
- **Data augmentation**
  - Geometric transformations (on images)
  - Adding noise
    - Dropout

Image source: https://github.com/aleju/imgaug

Last week - Avoiding overfitting - Data augmentation

Data augmentation

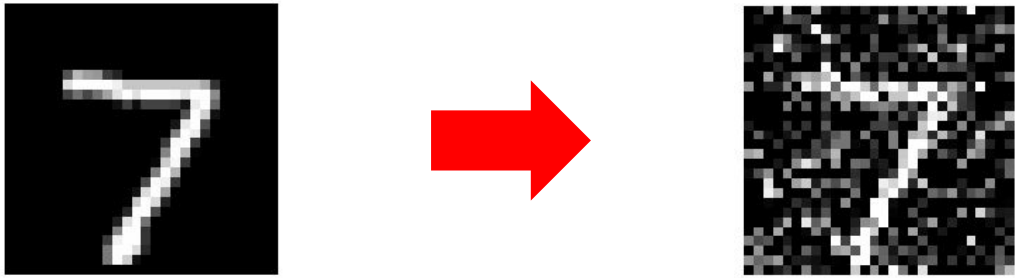


Obtaining more training data is often not feasible.
→ **Instead, create "new" training data by applying transformations on existing data.**

Last week - Avoiding overfitting - Noise

Adding noise (A type of data augmentation)

Adding Gaussian noise to the input can reduce overfitting.



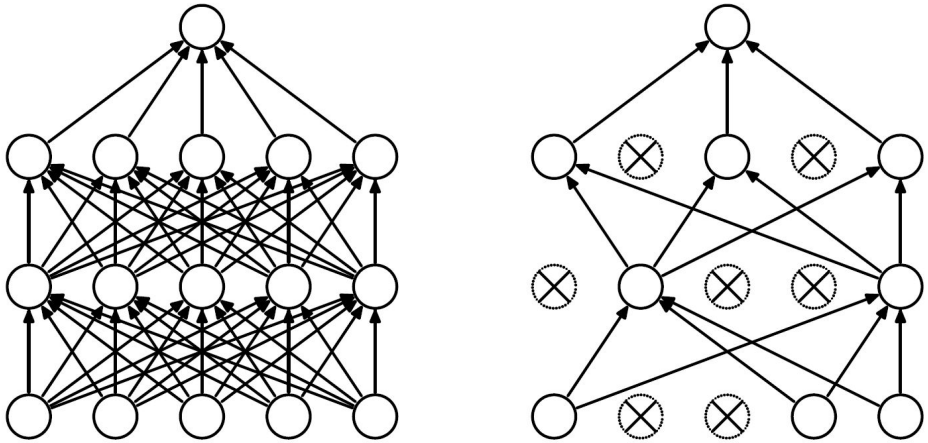
Adding noise prevents the network from “memorizing” precise details, thereby mitigating overfitting.

Image source: Srivastava et al., 2014

Last week - Avoiding overfitting - Dropout

Dropout (A type of noise)

We randomly set some elements of the hidden representations to zero.



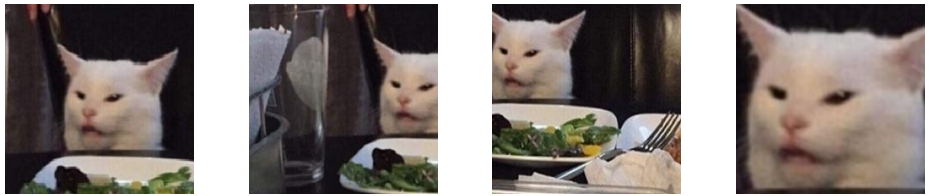
(a) Standard Neural Net

(b) After applying dropout.

Last week - MLP is not ideal for image processing

Multilayer Perceptron

- **Ideal when an input variable carries a “fixed, distinct meaning”**
  (This is typically not the case for image pixels)
- **Lack of translation invariance / equivariance**
  (Pattern recognition in the image is location-dependent)
- **Ignores the neighborhood relationships between input variables**



Last week - 2D Convolution, discrete case

$$\hat{y}[i,j] = (x * w)[i,j] = \langle x[i..i+U, j..j+V] \, , \, w \rangle =$$

$$= \sum_{u=1}^{U} \sum_{v=1}^{V} x[i+u, j+v] \cdot w[u,v]$$

1	2	2	-2
1	0	2	4
2	-1	-1	3
0	0	-3	-2

x

*

1	0	3
2	-1	-1
0	1	-3

w

=

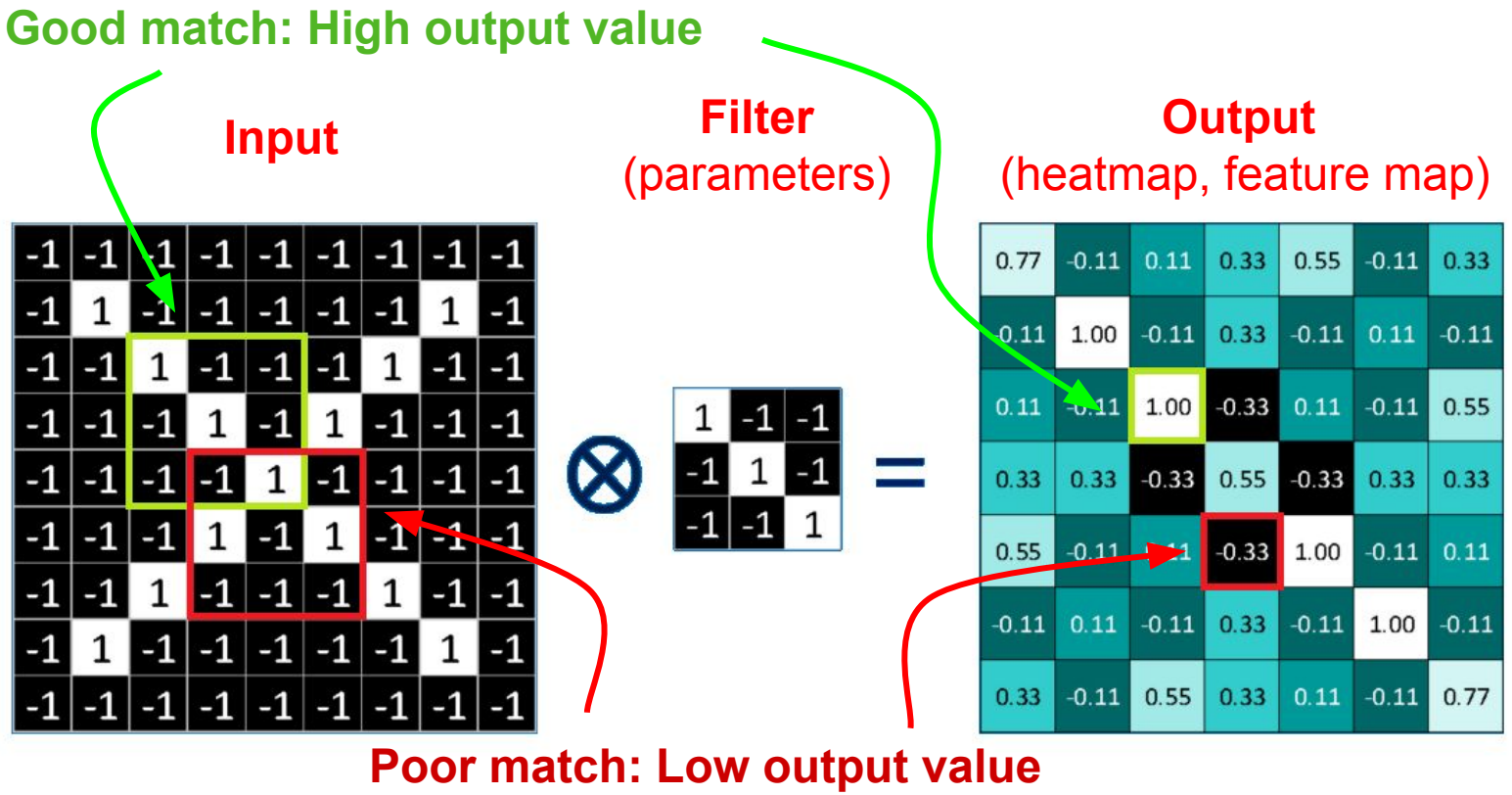
9	-20
22	11

y

$$0 \cdot 1 + 2 \cdot 0 + 4 \cdot 3 + (-1) \cdot 2 + (-1) \cdot (-1) + 3 \cdot (-1) + 0 \cdot 0 + (-3) \cdot 1 + (-2) \cdot (-3) = 11$$

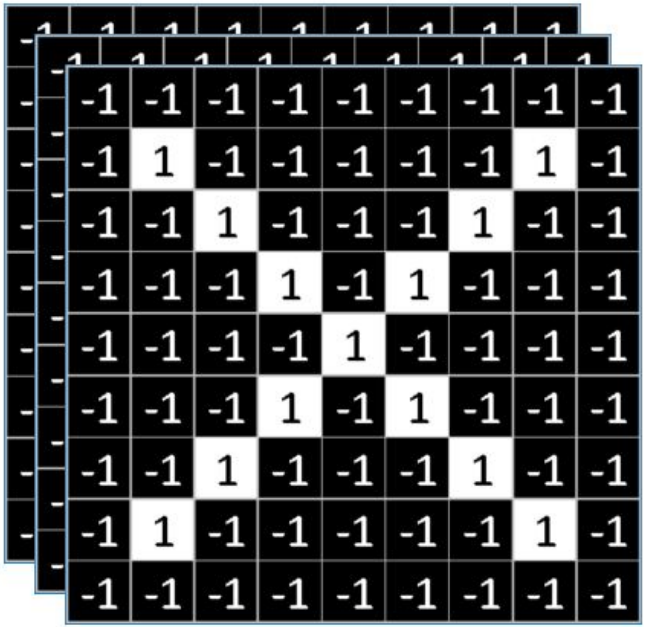
Image source: https://brohrer.github.io/how_convolutional_neural_networks_work.html

Last week - Convolution for pattern recognition



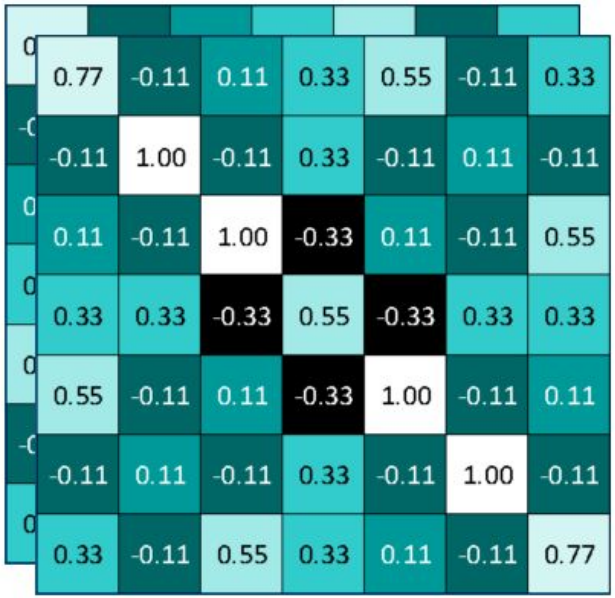
Last week - Convolutional layer - Channels

P input channels



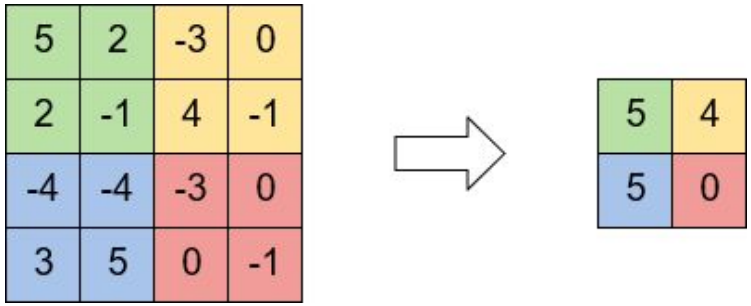
Convolutional layer
Q filters
(each filter with **P** channels).

Q output channels

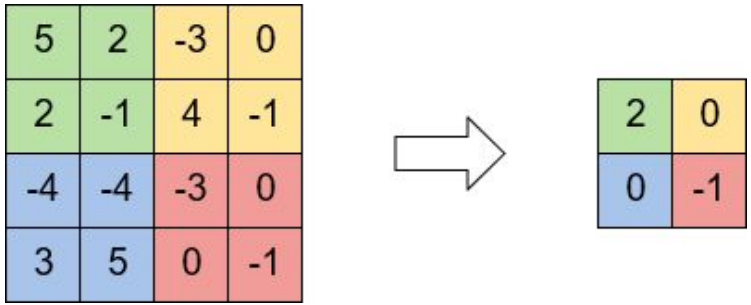


Last week - Downsampling / pooling layer

Max pooling (2x2)



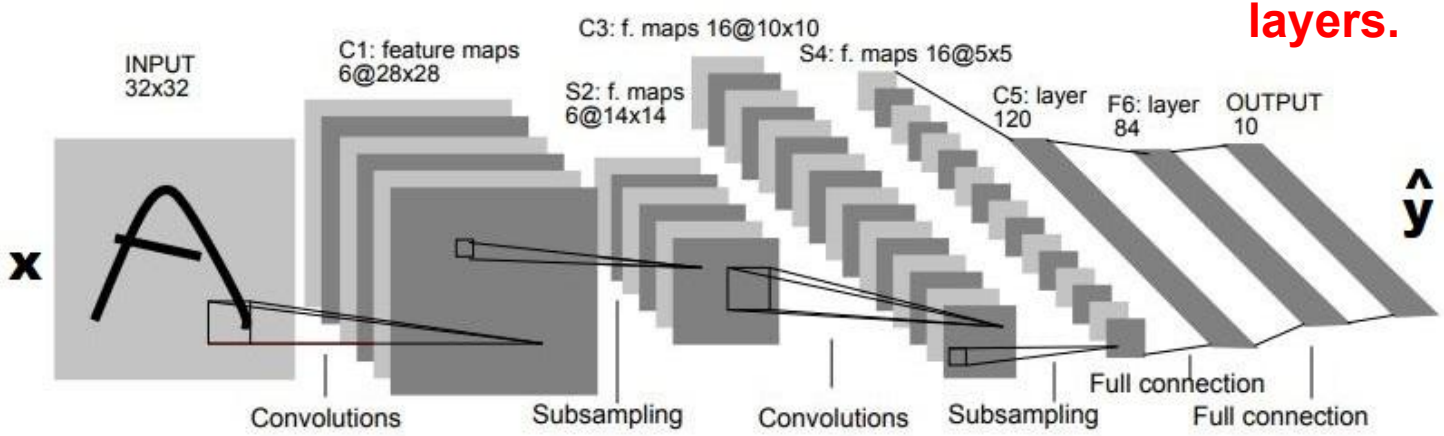
Average pooling (2x2)



Last week - Convolutional network (LeNet-5)

Convolutional and pooling layers in an alternating fashion.

Finally, fully connected layers.



Last week - Convolutional network

Alternating convolution and pooling layers
→ **Hierarchical pattern recognition**

The network learns to recognize larger, higher-level patterns as combinations of several smaller, simpler patterns.

Previously - PyTorch MLP

```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, n_categories):
      super().__init__()
      self.fc1 = nn.Linear(input_dim, 20)
      self.fc2 = nn.Linear(20, n_categories)

  def forward(self, x):
      # Expecting x.shape == (batch_size, input_dim)
      out = self.fc1(x)
      out = nn.ReLU()(out)
      out = self.fc2(out)
      return out
```

Definition,
initialization of layers/parameters.

Computing the output from the input
(feed-forward)

Previously - PyTorch MLP

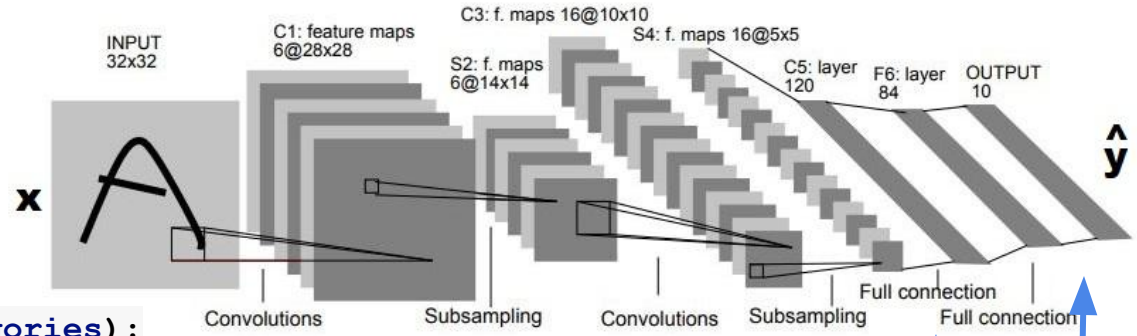
```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, n_categories):
      super().__init__()
      self.fc1 = nn.Linear(input_dim, 20)
      self.fc2 = nn.Linear(20, n_categories)

  def forward(self, x):
      # Expecting x.shape == (batch_size, input_dim)
      out = self.fc1(x)
      out = nn.ReLU()(out)
      out = self.fc2(out)
      return out
```

Definition,
initialization of layers/parameters.

(batch_size, input_dim)
(batch_size, 20)
(batch_size, 20)
(batch_size, 10)

Computing the output from the input
(feed-forward)

PyTorch CNN




```
class LeNet5(nn.Module):
    def __init__(self, n_categories):
        super().__init__()
        self.conv_block1 = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size = (5,5)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = (2,2)))
        self.conv_block2 = nn.Sequential(
            nn.Conv2d(6, 16, kernel_size = (5,5)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = (2,2)))
        self.fc1 = nn.Linear(400, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, n_categories)
```

PyTorch CNN

Definition, initialization of layers/parameters.

```
class LeNet5(nn.Module):
    def __init__(self, n_categories):
        super().__init__()
        self.conv_block1 = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size = (5,5)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = (2,2)))
        self.conv_block2 = nn.Sequential(
            nn.Conv2d(6, 16, kernel_size = (5,5)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = (2,2)))
        self.fc1 = nn.Linear(400, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, n_categories)
```

Computing the output from the input (feed-forward)

```
def forward(self, x):
  # x.shape == (batch_size, 1, 32, 32)
  out = self.conv_block1(x)
  out = self.conv_block2(out)
  out = out.reshape(out.size(0), -1)
  out = self.fc1(out)
  out = nn.ReLU()(out)
  out = self.fc2(out)
  out = nn.ReLU()(out)
  out = self.fc3(out)
  return out
```

PyTorch CNN

Conv2d **MaxPool2d**

(bs, 1, 32, 32) → (bs, 6, 28, 28) → (bs, 6, 14, 14)

```
class LeNet5(nn.Module):
  def __init__(self, n_categories):
    super().__init__()
    self.conv_block1 = nn.Sequential(
      nn.Conv2d(1, 6, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.conv_block2 = nn.Sequential(
      nn.Conv2d(6, 16, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.fc1 = nn.Linear(400, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, n_categories)
```

```
def forward(self, x):
  # x.shape == (batch_size, 1, 32, 32)
  out = self.conv_block1(x)
  out = self.conv_block2(out)
  out = out.reshape(out.size(0), -1)
  out = self.fc1(out)
  out = nn.ReLU()(out)
  out = self.fc2(out)
  out = nn.ReLU()(out)
  out = self.fc3(out)
  return out
```

(bs, 1, 32, 32)

(bs, 6, 14, 14)

PyTorch CNN

Conv2d **MaxPool2d**

(bs, 6, 14, 14) → (bs, 16, 10, 10) → (bs, 16, 5, 5)

```
class LeNet5(nn.Module):
  def __init__(self, n_categories):
    super().__init__()
    self.conv_block1 = nn.Sequential(
      nn.Conv2d(1, 6, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.conv_block2 = nn.Sequential(
      nn.Conv2d(6, 16, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.fc1 = nn.Linear(400, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, n_categories)
```

```
def forward(self, x):
  # x.shape == (batch_size, 1, 32, 32)
  out = self.conv_block1(x)
  out = self.conv_block2(out)
  out = out.reshape(out.size(0), -1)
  out = self.fc1(out)
  out = nn.ReLU()(out)
  out = self.fc2(out)
  out = nn.ReLU()(out)
  out = self.fc3(out)
  return out
```

(bs, 1, 32, 32)

(bs, 6, 14, 14)

(bs, 16, 5, 5)

PyTorch CNN

```
class LeNet5(nn.Module):
  def __init__(self, n_categories):
    super().__init__()
    self.conv_block1 = nn.Sequential(
      nn.Conv2d(1, 6, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.conv_block2 = nn.Sequential(
      nn.Conv2d(6, 16, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.fc1 = nn.Linear(400, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, n_categories)
```

```
def forward(self, x):
  # x.shape == (batch_size, 1, 32, 32)
  out = self.conv_block1(x)
  out = self.conv_block2(out)
  out = out.reshape(out.size(0), -1)
  out = self.fc1(out)
  out = nn.ReLU()(out)
  out = self.fc2(out)
  out = nn.ReLU()(out)
  out = self.fc3(out)
  return out
```

(bs, 1, 32, 32)
(bs, 6, 14, 14)
(bs, 16, 5, 5)
(bs, 400)
(bs, 120)
(bs, 120)
(bs, 84)
(bs, 84)
(bs, 10)

PyTorch CNN

```
class LeNet5(nn.Module):
  def __init__(self, n_categories):
    super().__init__()
    self.conv_block1 = nn.Sequential(
      nn.Conv2d(1, 6, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.conv_block2 = nn.Sequential(
      nn.Conv2d(6, 16, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.fc1 = nn.Linear(400, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, n_categories)
```

```
def forward(self, x):
  # x.shape == (batch_size, 1, 32, 32)
  out = self.conv_block1(x)
  out = self.conv_block2(out)
  out = out.reshape(out.size(0), -1)
  out = self.fc1(out)
  out = nn.ReLU()(out)
  out = self.fc2(out)
  out = nn.ReLU()(out)
  out = self.fc3(out)
  return out
```

(bs, 1, 32, 32)
(bs, 6, 14, 14)
(bs, 16, 5, 5)
(bs, 400)
(bs, 120)
(bs, 120)
(bs, 84)
(bs, 84)
(bs, 10)

Last activation function and loss function chosen depending on the task…
(regression, binary classification, multi-class classification)

Image source: http://datahacker.rs/what-is-padding-cnn/

Convolutional network - hyperparameters

Padding:

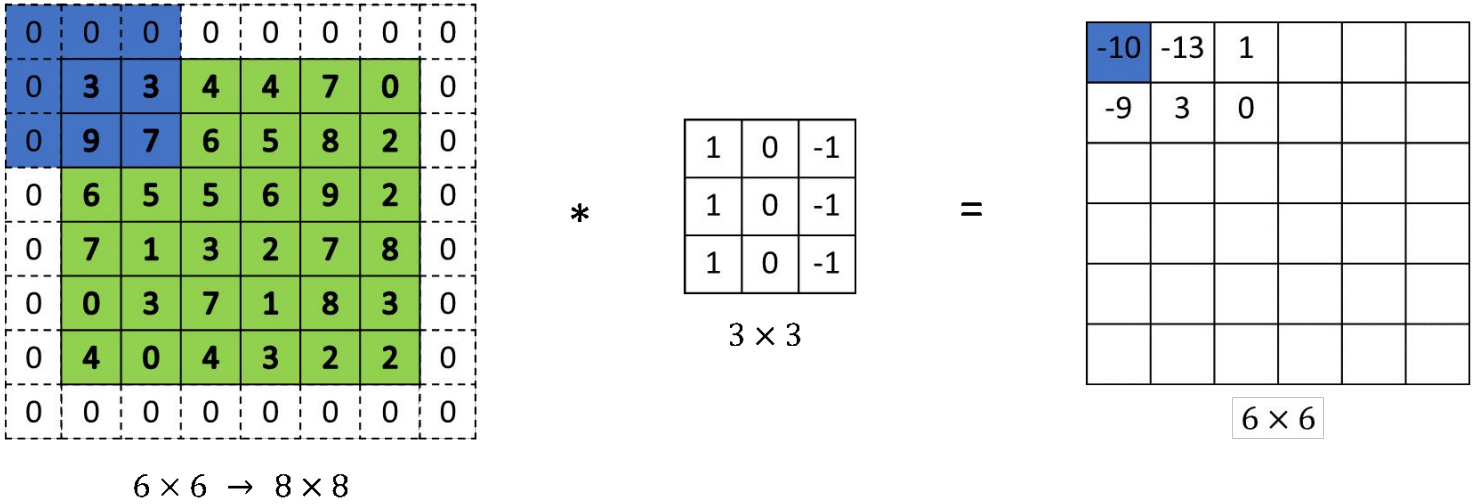
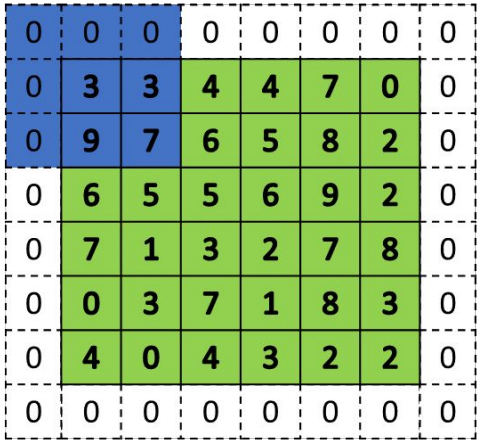


Image source: http://datahacker.rs/what-is-padding-cnn/

Convolutional network - hyperparameters

Padding:

```
nn.Conv2d(..., padding=1, ...)
nn.Conv2d(..., padding="same", ...)
```



$6 \times 6 \rightarrow 8 \times 8$

$*$

1	0	-1
1	0	-1
1	0	-1

3×3

$=$

-10	-13	1			
-9	3	0			

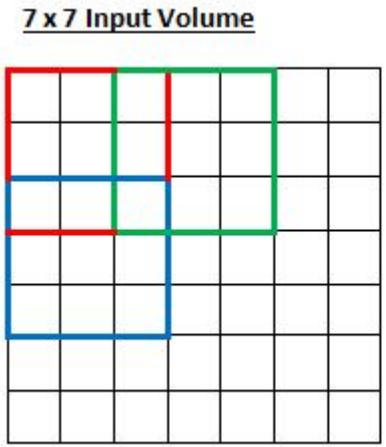
6×6

Padding the input at the edges (e.g., with zeros),
to change the size of the output (e.g., to have the same output size as the input)

Image source: https://adeshpande3.github.io/

Convolutional network - hyperparameters

Stride (step size)**:**



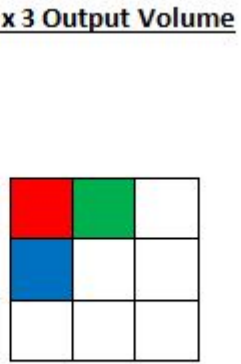
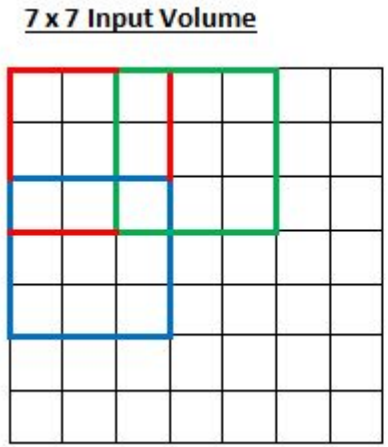


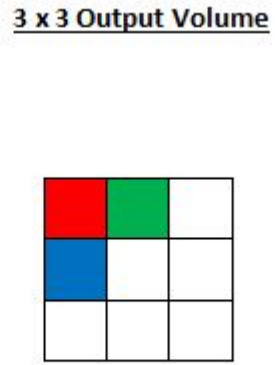
Image source: https://adeshpande3.github.io/

Convolutional network - hyperparameters

Stride (step size):

```
nn.Conv2D(..., stride=2, ...)
nn.MaxPooling2D(..., stride=(4, 3), ...)
```





We can also apply convolution to the input with a larger **stride**, which **results in a lower-resolution image**… The stride of pooling layers can also vary.

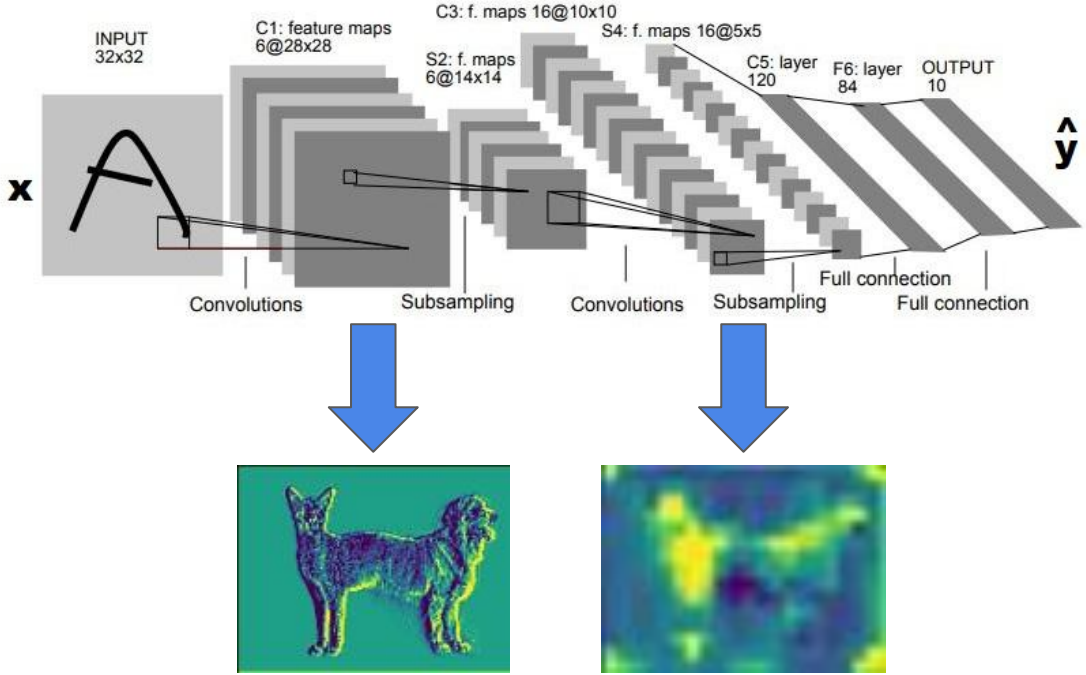
What does a Convolutional network learn?

Visualizing heatmaps - Jupyter notebook on Google Colab

https://colab.research.google.com/drive/1I_Uv_QMIr2fn_B7eHJnPgtuyFelmiU_W

What does a Convolutional network learn?

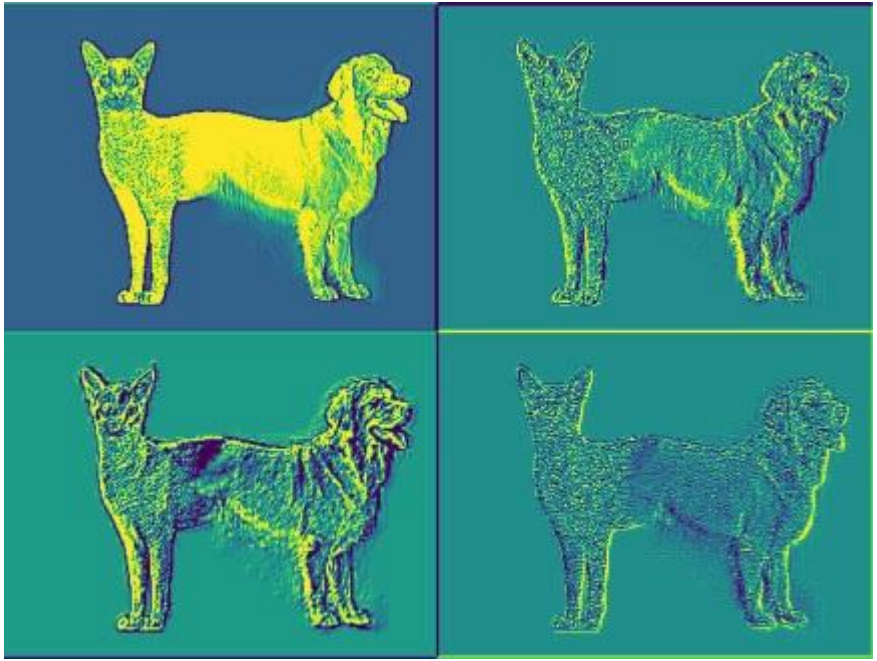
Visualizing heatmaps



What does a Convolutional network learn?

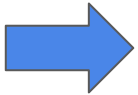
Visualizing heatmaps - First layer output

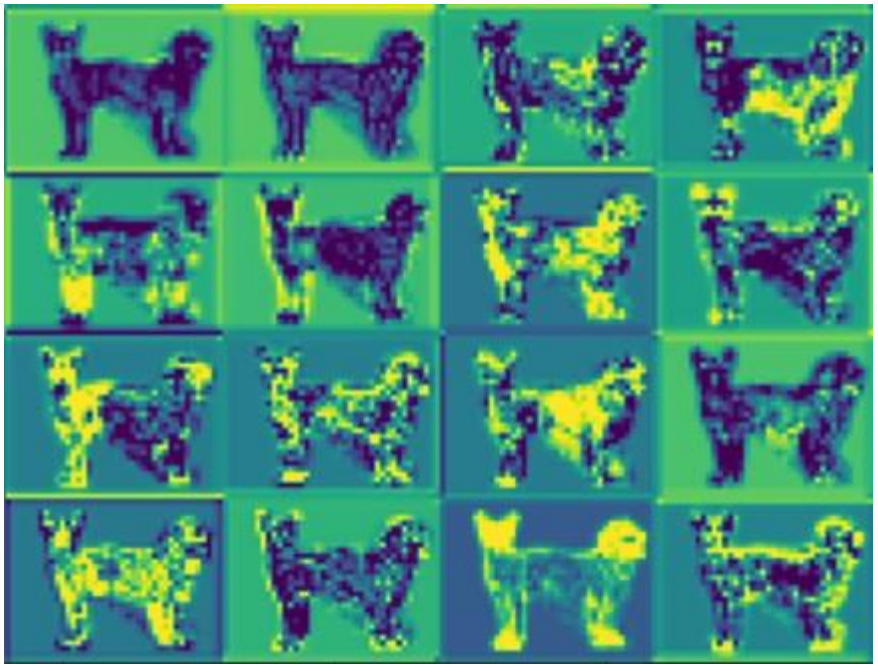
Heatmaps produced by a deep ConvNet trained on the ImageNet dataset…



What does a Convolutional network learn?

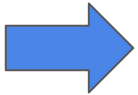
Visualizing heatmaps - Middle layer output

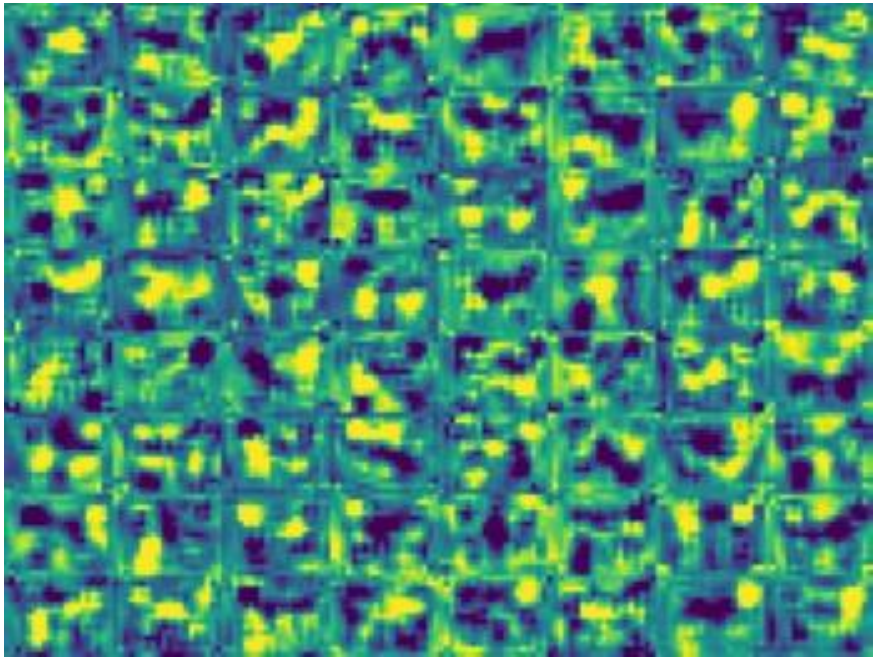




What does a Convolutional network learn?

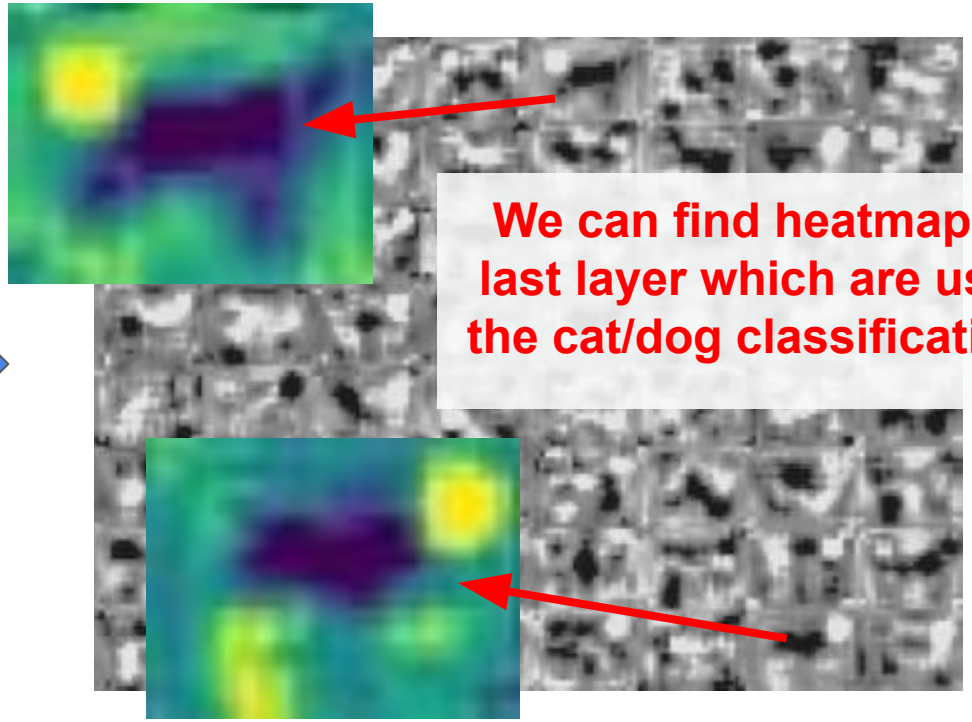
Visualizing heatmaps - Last layer output





What does a Convolutional network learn?

Visualizing heatmaps - Last layer output



We can find heatmaps in the last layer which are useful for the cat/dog classification task!

Image source: Yosinski et al. 2015, Deep Visualization Toolbox

What does a Convolutional network learn?

Visualizing heatmaps

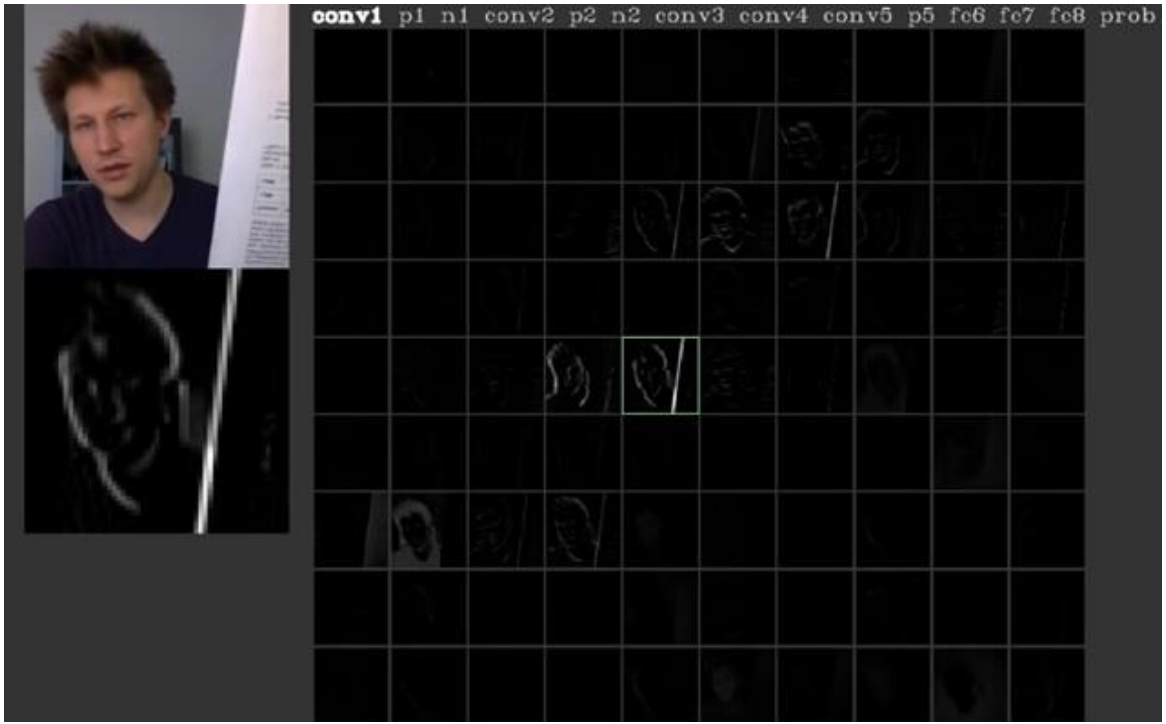


Image source: Yosinski et al. 2015, Deep Visualization Toolbox

What does a Convolutional network learn?

Visualizing heatmaps

A deep neural network trained on ImageNet (AlexNet, 2012)

The first layer is responsible for low-level patterns:

One of the filters learned the vertical, dark-to-light transition

Input **Heatmaps from the first conv. layer**

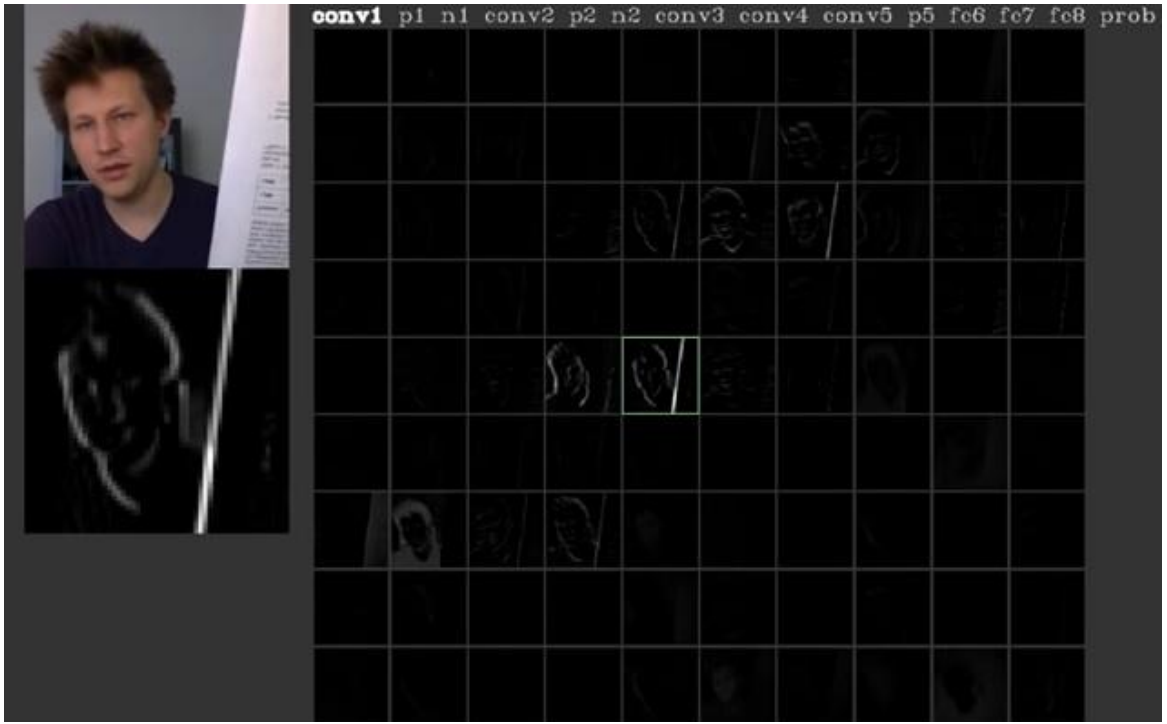


Image source: Yosinski et al. 2015, Deep Visualization Toolbox

What does a Convolutional network learn?

Visualizing heatmaps

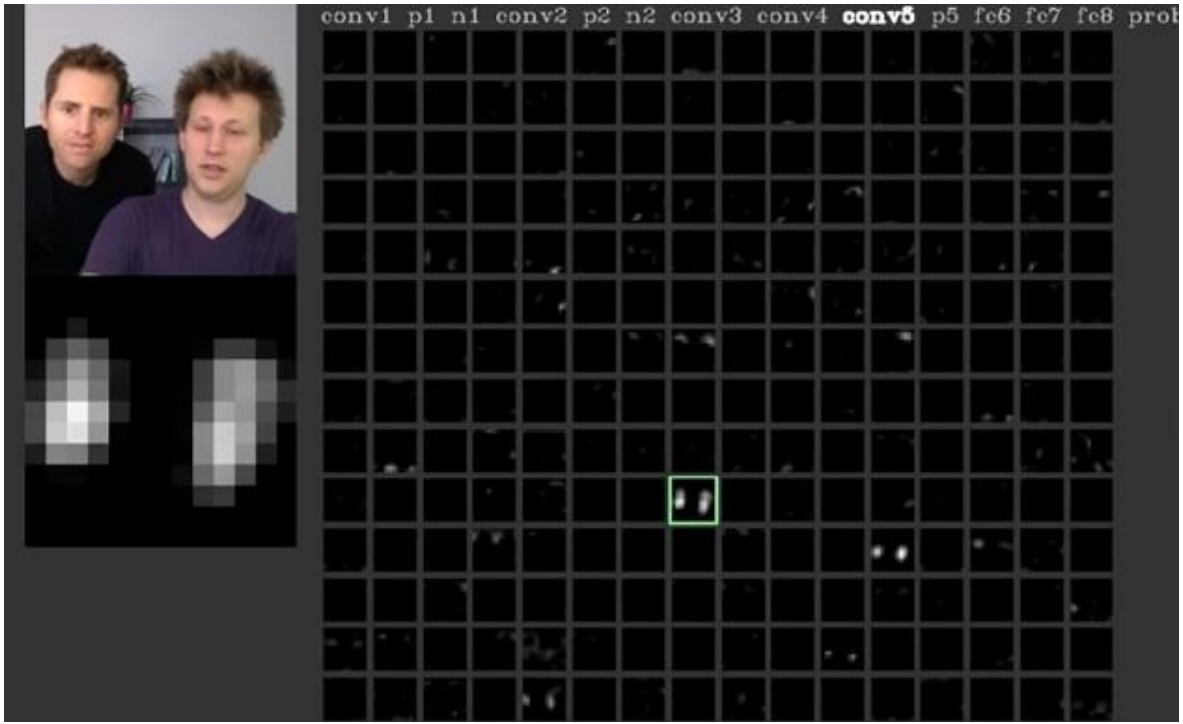


Image source: Yosinski et al. 2015, Deep Visualization Toolbox

What does a Convolutional network learn?

Visualizing heatmaps

A deep neural network trained on ImageNet (AlexNet, 2012)

The last layers are responsible for high-level patterns:

One filter learned to recognize faces.

Input **Heatmaps from the fifth conv. layer**

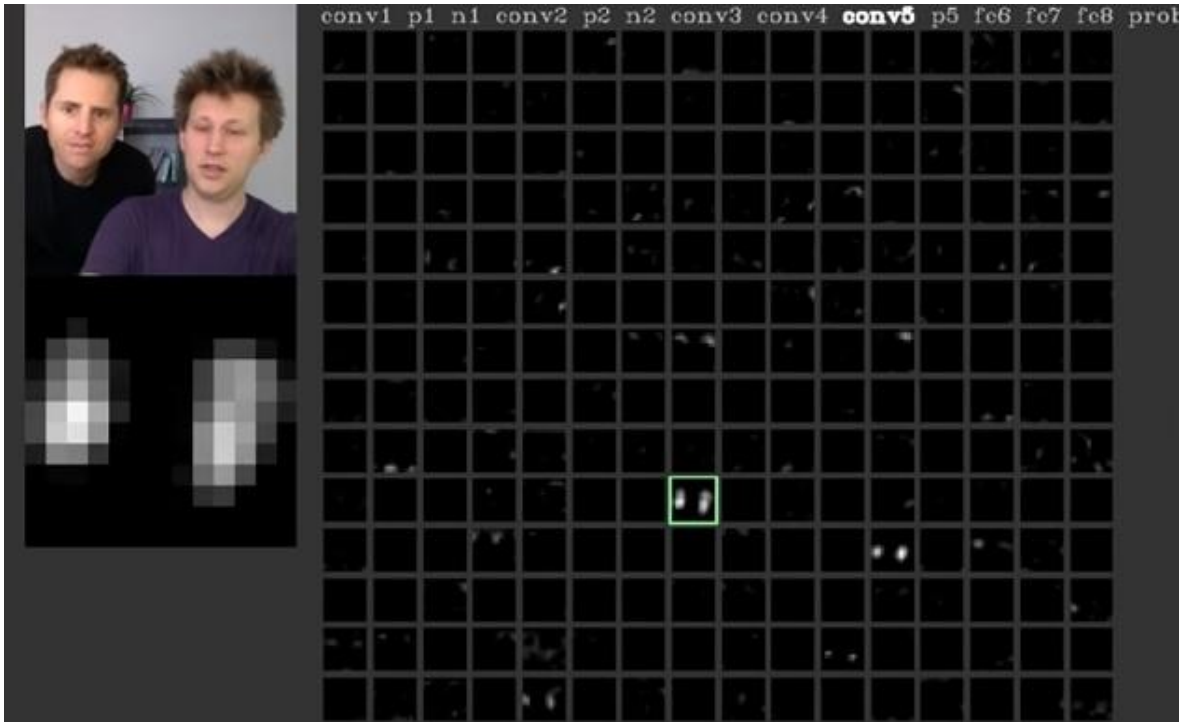


Image source: Yosinski et al. 2015, Deep Visualization Toolbox

What does a Convolutional network learn?

Visualizing heatmaps

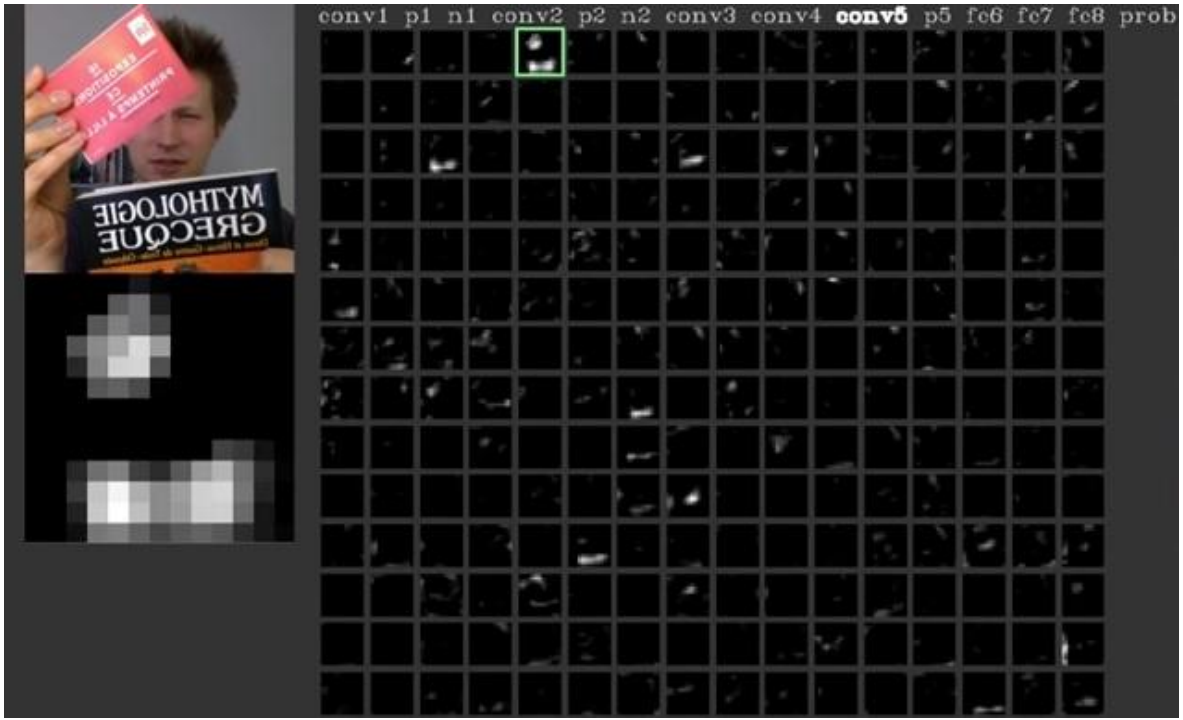


Image source: Yosinski et al. 2015, Deep Visualization Toolbox

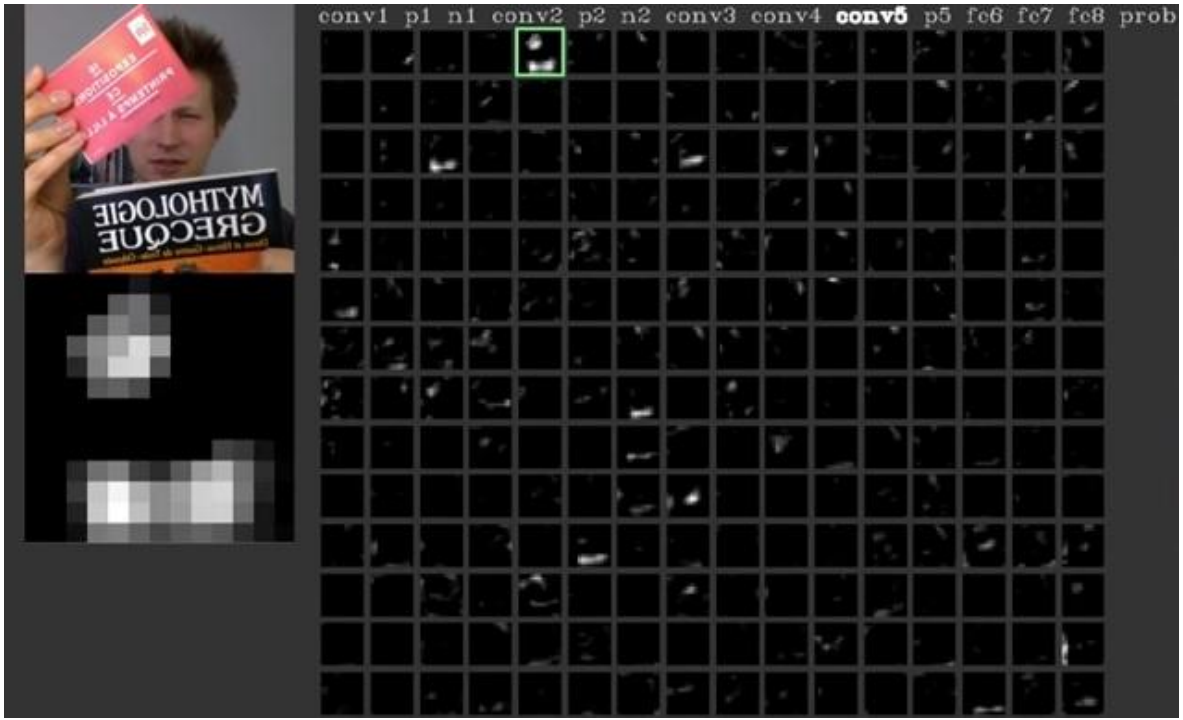
What does a Convolutional network learn?

Visualizing heatmaps

A deep neural network trained on ImageNet (AlexNet, 2012)

The last layers are responsible for high-level patterns:

Another filter learned to recognize printed text.



What does a Convolutional network learn?

Let's turn it around: Let's look for inputs that produce high values on individual heatmaps!

Image source: Zeiler et al. 2013, https://arxiv.org/pdf/1311.2901.pdf

What does a Convolutional network learn?

Let's turn it around: Let's look for inputs that produce high values on individual heatmaps!

First conv. layer

Middle conv. layer

Last conv. layer

Image source: Zeiler et al. 2013, https://arxiv.org/pdf/1311.2901.pdf

What does a Convolutional network learn?

Hierarchy: The layers, which build upon one another, learn increasingly complex patterns.

Filter #1 of the first layer matches this kind of input the best…

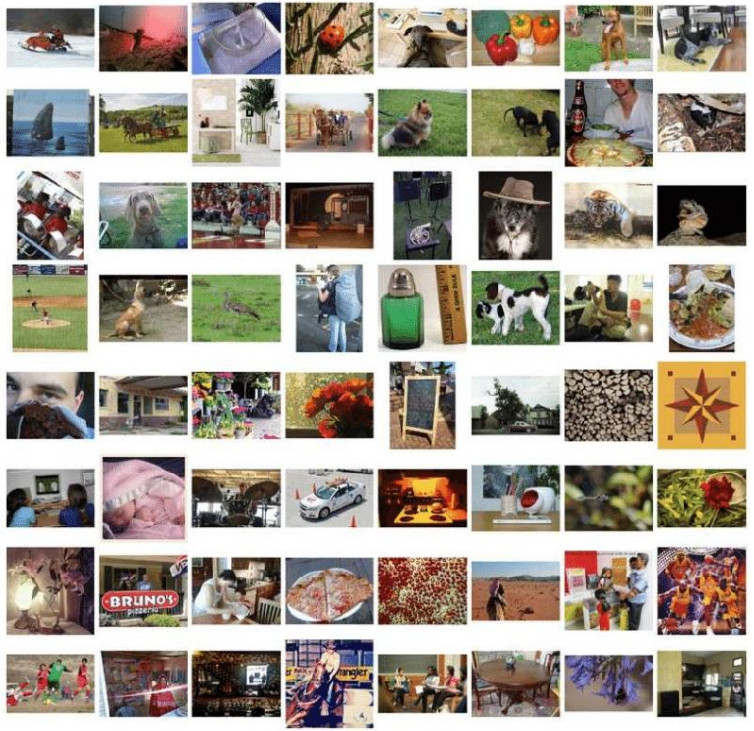
… filter #2

First conv. layer **Middle conv. layer** **Last conv. layer**

Previously - ImageNet dataset

ImageNet - Photos of various object categories.

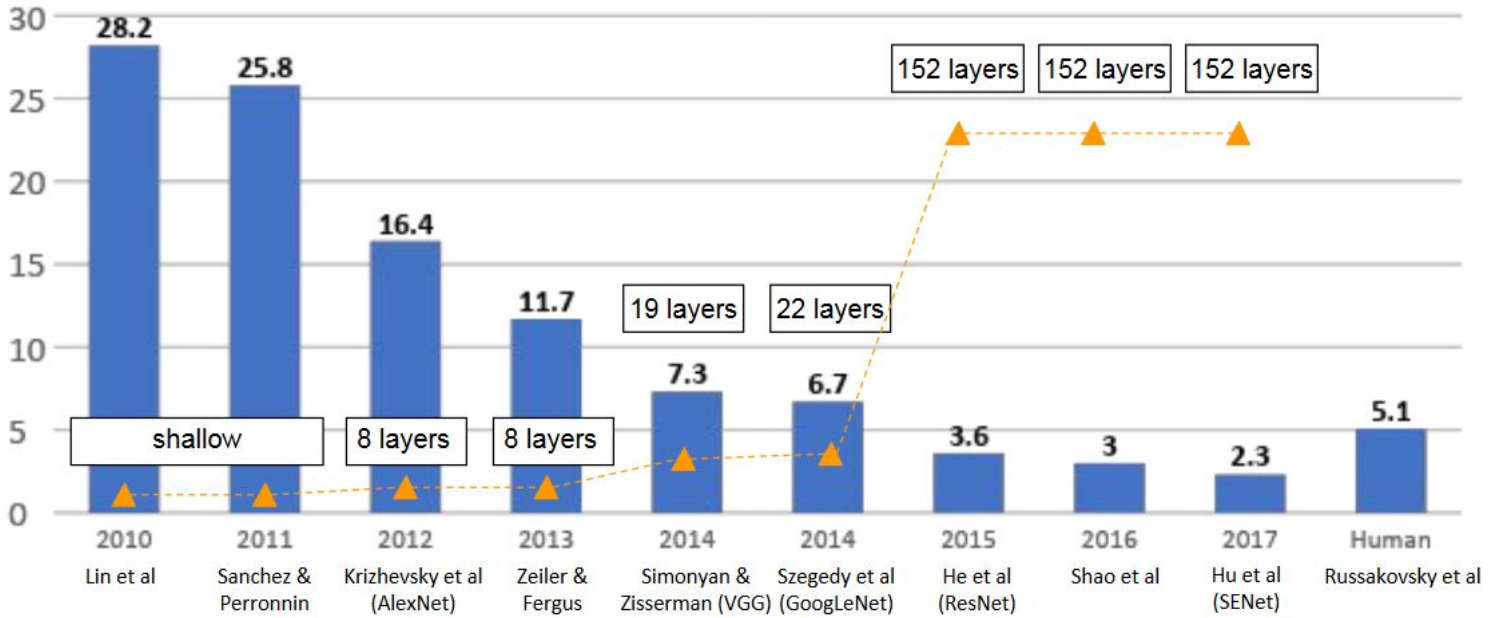
- Over 1,000 categories, such as:
  - swamp turtle
  - waffle iron
  - Norfolk terrier
  - viaduct
- Around 1 megapixel, color
  → **approx. 3 million input variables**



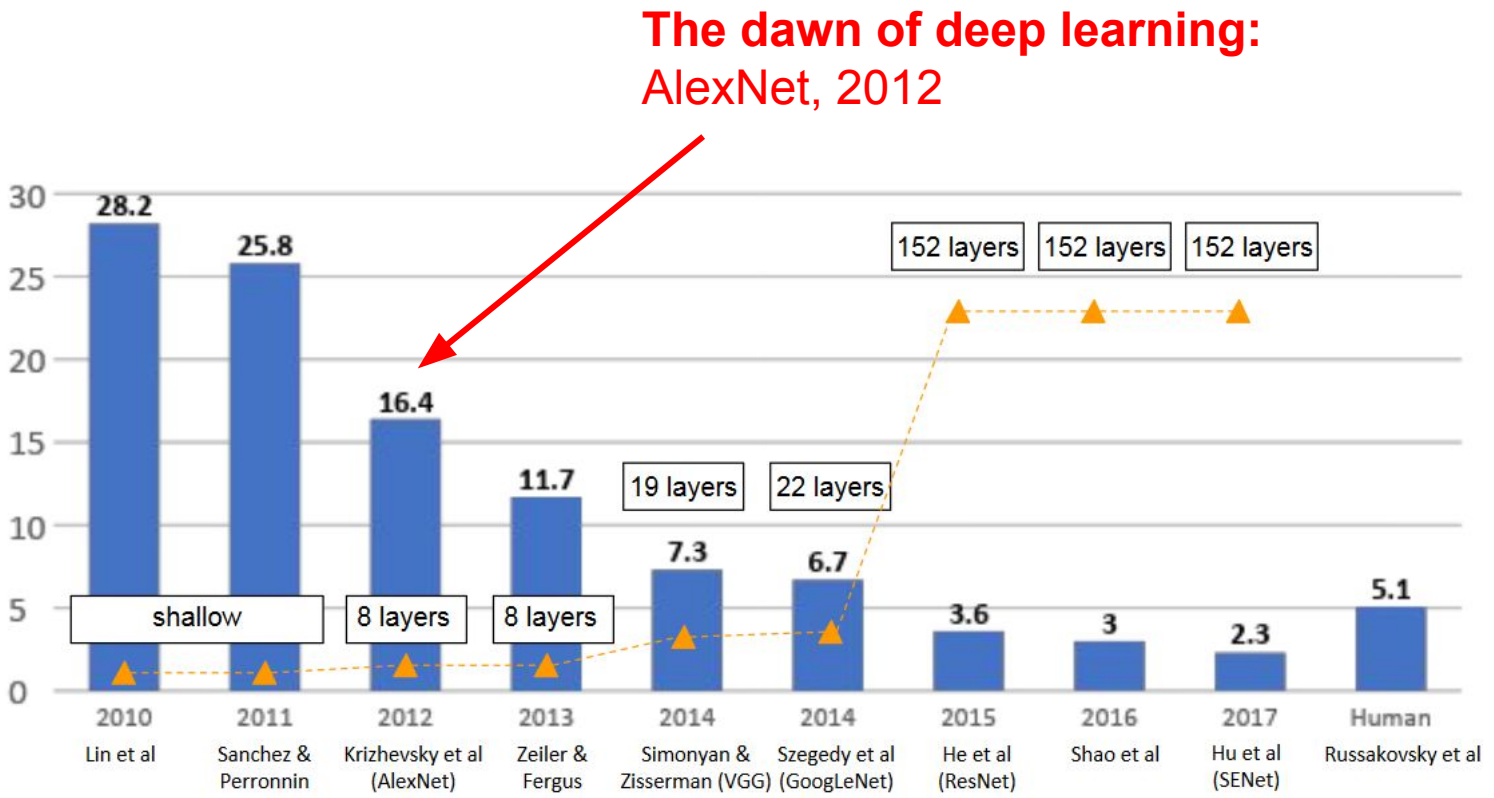
Previously - ImageNet dataset

ImageNet ILSVRC Challenge Winners - every year

(1,000-class image classification, 1-class accuracy over 5 trials)



Previously - ImageNet dataset



Previously - ImageNet dataset

A highly popular deep network from 2014: VGG-19

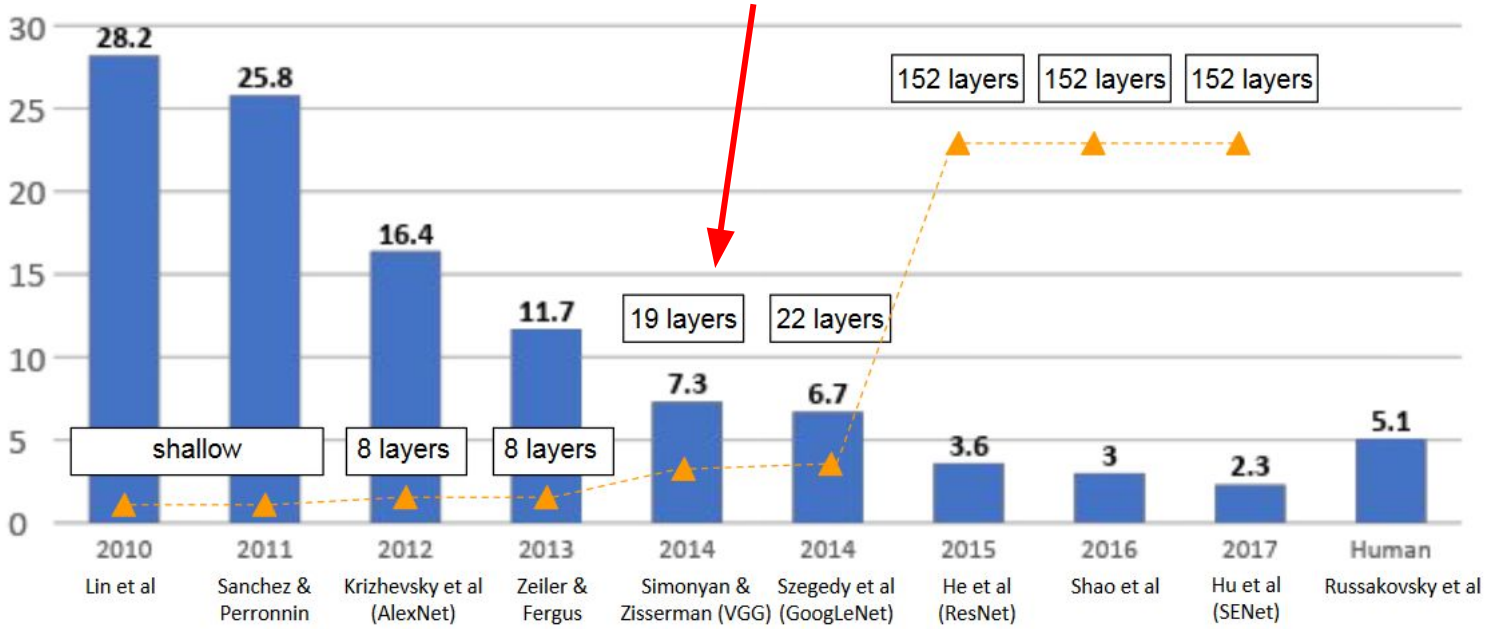


Image source: Simonyan & Zisserman 2014

VGG-19

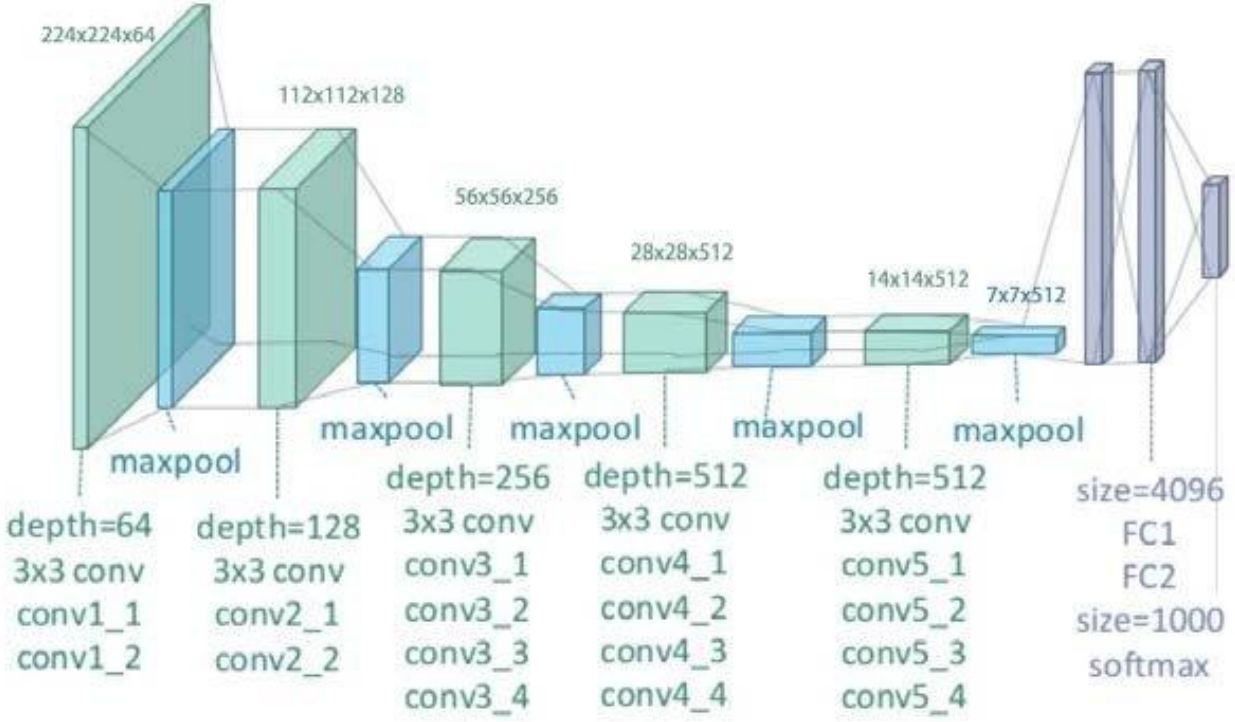


Image source: Simonyan & Zisserman 2014

VGG-19

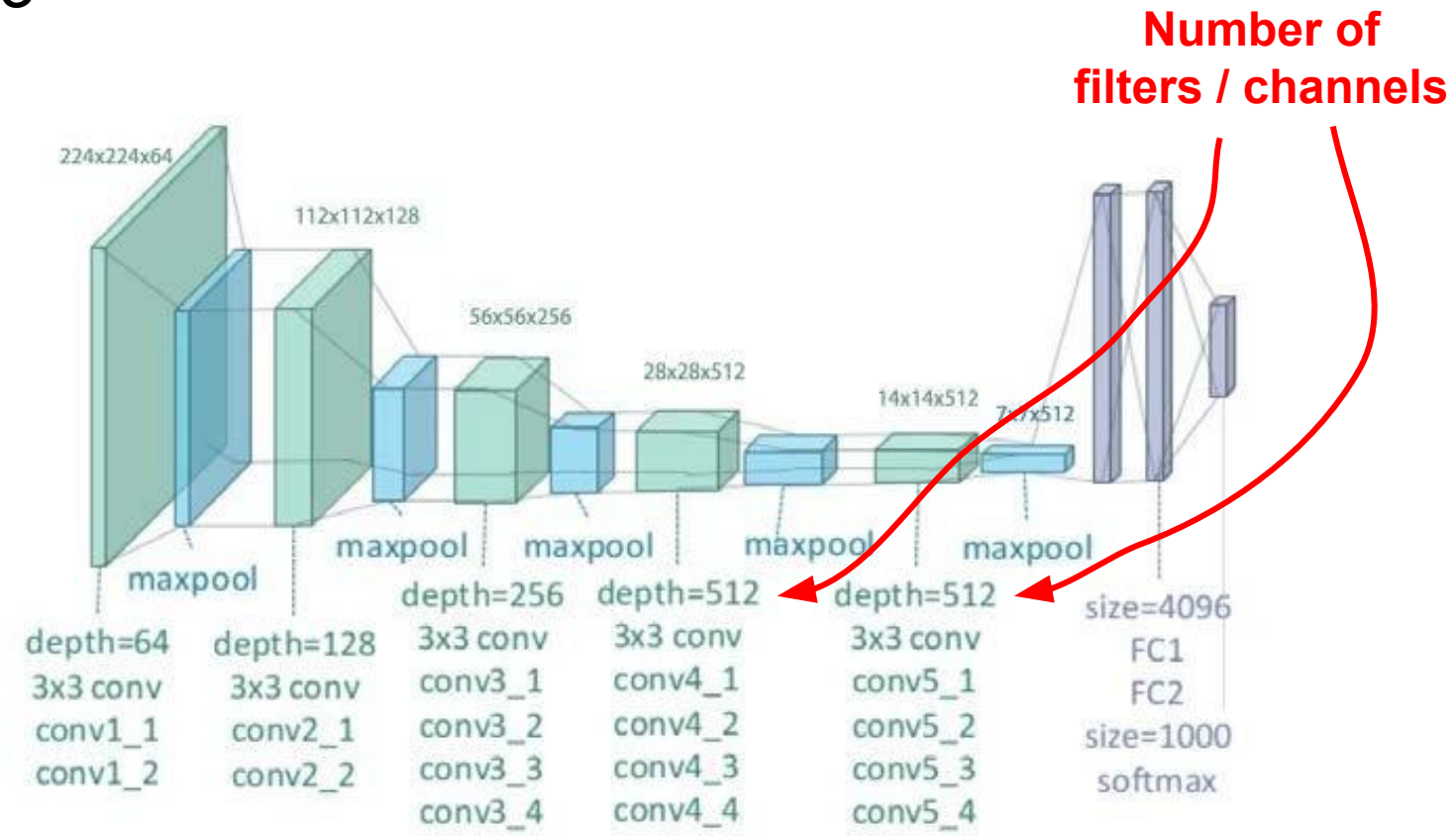
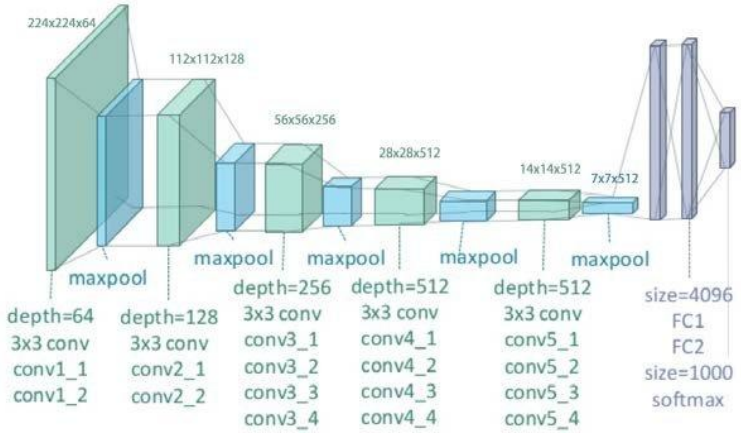


Image source: Simonyan & Zisserman 2014

VGG-19

- 92.7% top-5 accuracy (ILSVRC)
- 19 layers with parameters
- 143 million parameters
- One feed-forward: 20 GFLOPS
- Originally, it took 2–3 weeks
  to train on 4 high-end GPUs.



Deep learning - VGG

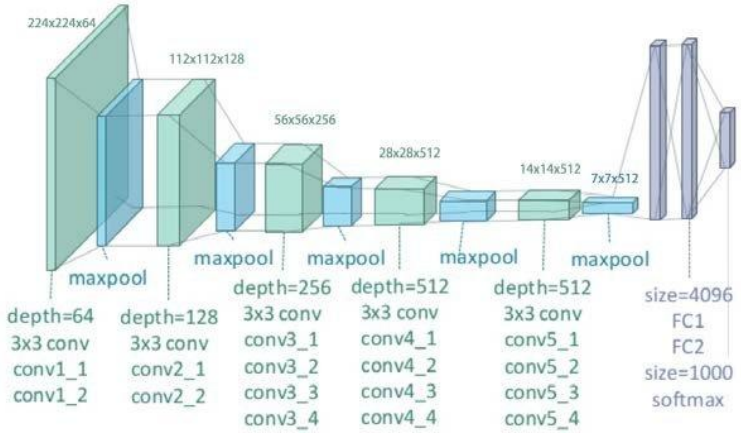
VGG-19 (2014) - Google search: *“VGG-19 training from scratch”*

GitHub issue:

Image source: Simonyan & Zisserman 2014

VGG-19

- 92.7% top-5 accuracy (ILSVRC)
- 19 layers with parameters
- 143 million parameters
- One feed-forward: 20 GFLOPS
- Originally, it took 2–3 weeks to train on 4 high-end GPUs.



sguada commented on Mar 2 2017

Training VGG_19 from scratch is very difficult.

Training VGG-19 (from scratch) is very difficult. **Why?**

The scaling of the gradient

Let’s take a fully connected layer as an example!

For convenience, let’s assume it consists of a single neuron with a single input, meaning that **w** is a 1×1 weight matrix, which we can interpret as a scalar...

$$\hat{y} = g(w \cdot x + b) \qquad w, b, x, \hat{y} \in \mathbb{R}$$

The scaling of the gradient

Let's examine **how the layer affects the gradient** back-propagated through it!

$$\hat{y} = g(wx + b) \qquad w, b, x, \hat{y} \in \mathbb{R}$$

$$\frac{\partial J}{\partial x} = \boxed{\frac{\partial \hat{y}}{\partial x}} \cdot \frac{\partial J}{\partial \hat{y}}$$

The derivative of the layer (as a function) (= the derivative of the output of the layer with respect to the input).

$$\frac{\partial \hat{y}}{\partial x} = \ ?$$

The scaling of the gradient

Let's examine **how the layer affects the gradient** back-propagated through it!

$$\hat{y} = g(wx + b) \qquad w, b, x, \hat{y} \in \mathbb{R}$$

$$\frac{\partial J}{\partial x} = \frac{\partial \hat{y}}{\partial x} \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

The scaling of the gradient

Let's examine **how the layer affects the gradient** back-propagated through it!

$$\hat{y} = g(wx + b) \qquad w, b, x, \hat{y} \in \mathbb{R}$$

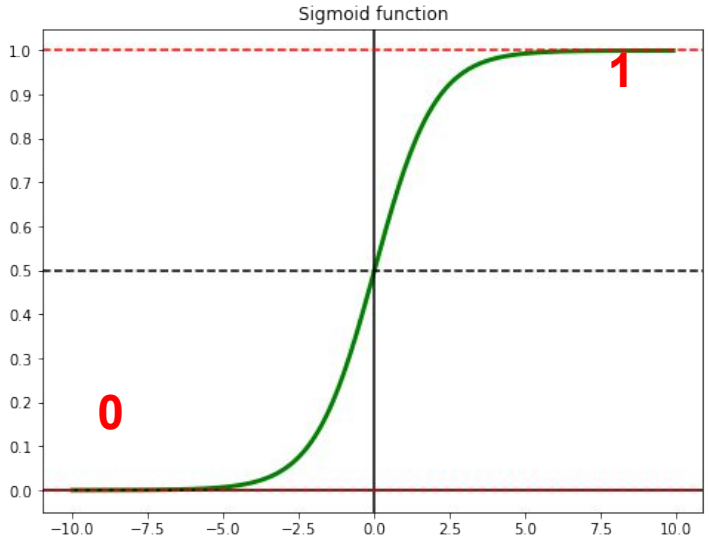
$$\frac{\partial J}{\partial x} = \boxed{\frac{\partial \hat{y}}{\partial x}} \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

Each layer **multiplies** the gradient **by the derivative of the activation function** and **the parameter w.**

The scaling of the gradient - Activation functions

Example: The sigmoid function



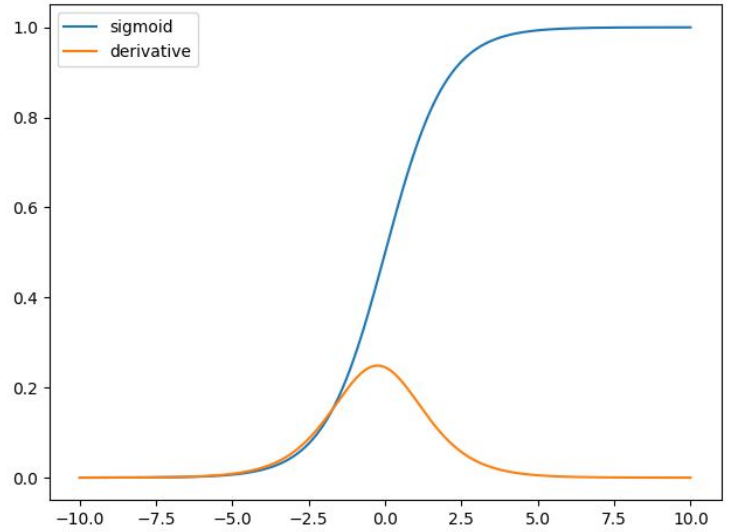
What is its derivative?

$$g(z) = \frac{1}{1+e^{-z}}$$

The scaling of the gradient - Activation functions

Example: The sigmoid function

Let's have a look at its derivative!



$$g(z) = \frac{1}{1+e^{-z}}$$

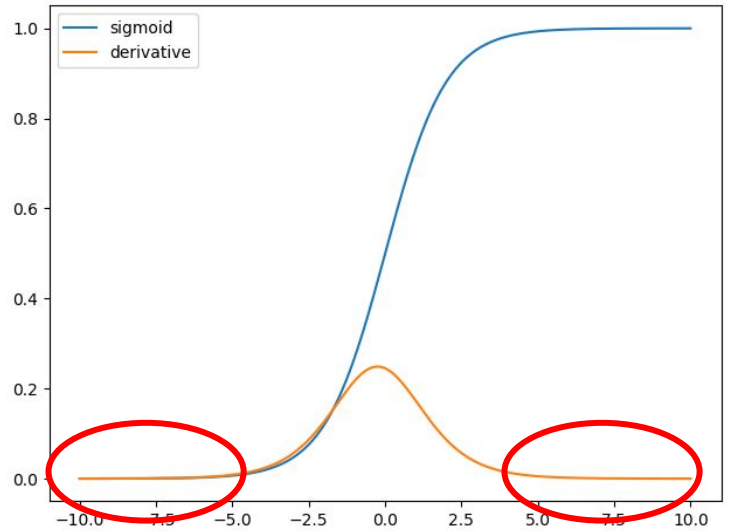
$$g'(z) = \ \ldots \ = g(z)(1-g(z))$$

The scaling of the gradient - Activation functions

Example: The sigmoid function

Let's have a look at its derivative!

$$g'(0) = 0.25$$
$$g'(-5) = g'(5) = 0.0066$$



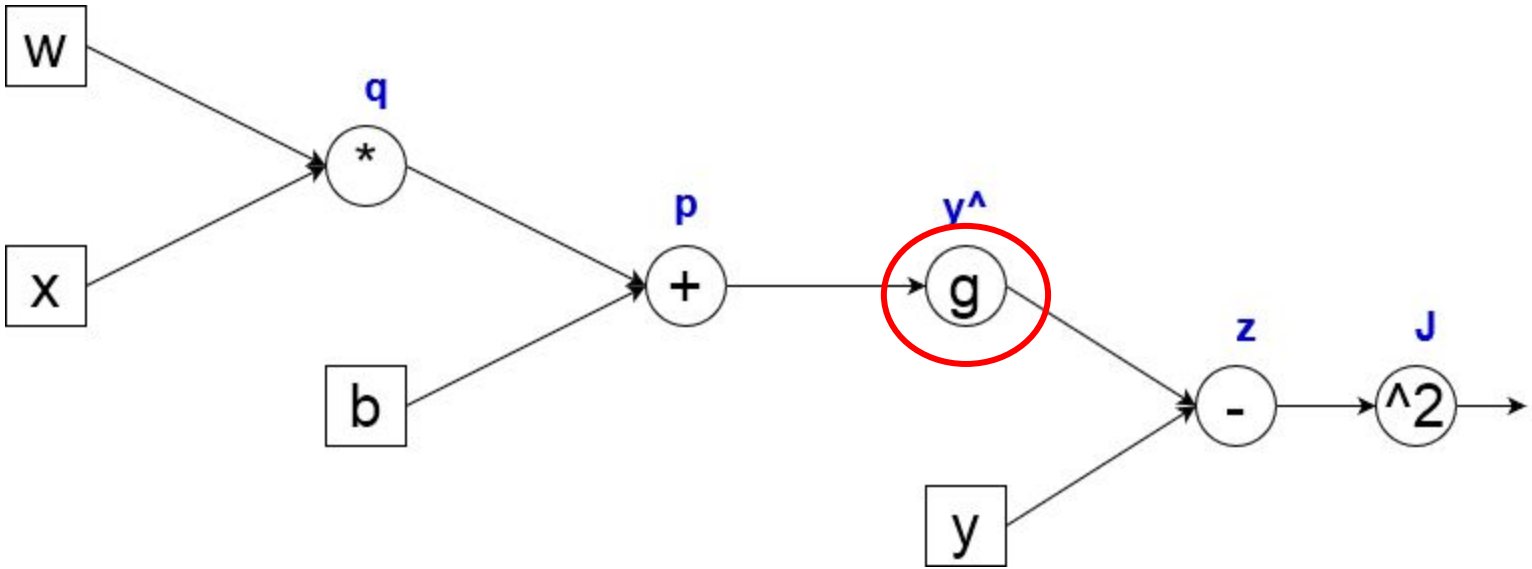
As the input moves away from zero, the derivative converges rapidly to zero.

$$g(z) = \frac{1}{1+e^{-z}}$$

$$g'(z) = \ \ldots \ = g(z)(1-g(z))$$

The scaling of the gradient

$$g'(0) = 0.25$$
$$g'(-5) = g'(5) = 0.0066$$

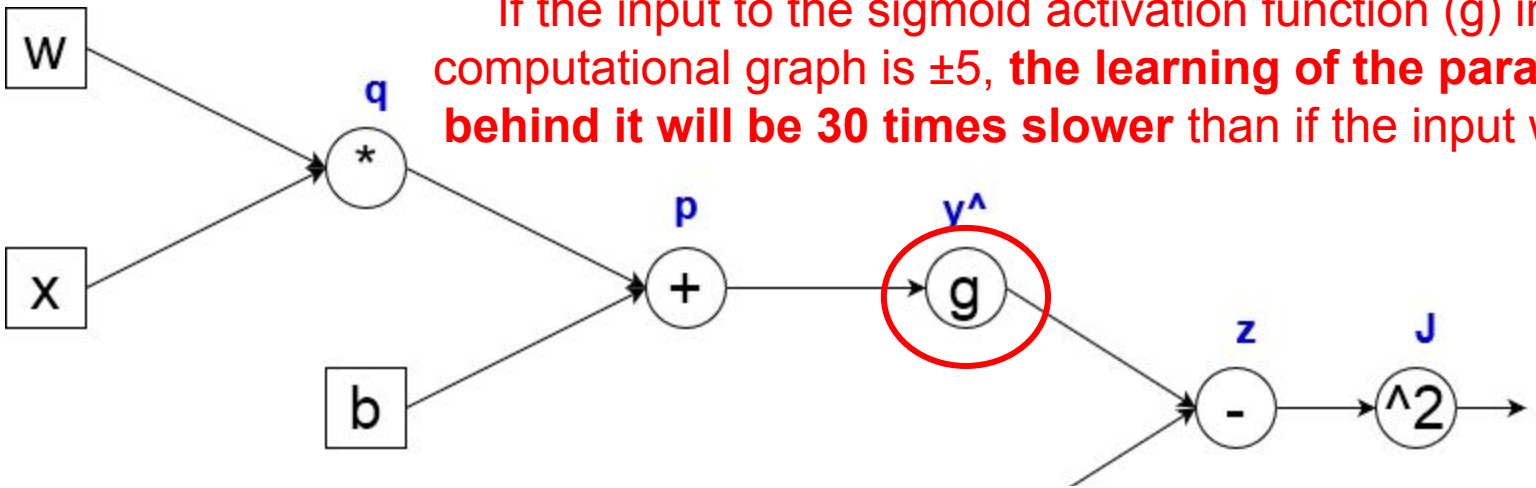


$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial z} \cdot \frac{\partial z}{\partial \hat{y}} \cdot \boxed{\frac{\partial \hat{y}}{\partial p}} \cdot \frac{\partial p}{\partial q} \cdot \frac{\partial q}{\partial w}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial z} \cdot \frac{\partial z}{\partial \hat{y}} \cdot \boxed{\frac{\partial \hat{y}}{\partial p}} \cdot \frac{\partial p}{\partial b}$$

The scaling of the gradient

$$g'(0) = 0.25$$
$$g'(-5) = g'(5) = 0.0066$$



If the input to the sigmoid activation function (g) in the computational graph is ±5, **the learning of the parameters behind it will be 30 times slower** than if the input were 0!

Multiple activation functions → Exponential effect!

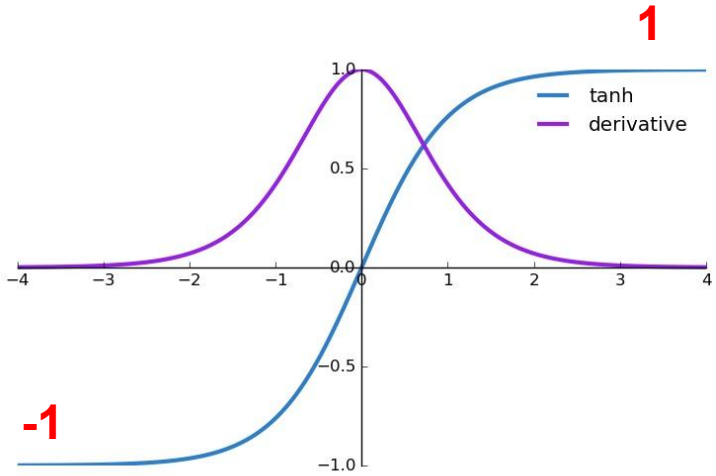
$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial z} \cdot \frac{\partial z}{\partial \hat{y}} \cdot \boxed{\frac{\partial \hat{y}}{\partial p}} \cdot \frac{\partial p}{\partial q} \cdot \frac{\partial q}{\partial w}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial z} \cdot \frac{\partial z}{\partial \hat{y}} \cdot \boxed{\frac{\partial \hat{y}}{\partial p}} \cdot \frac{\partial p}{\partial b}$$

Activation functions

Tanh (Hyperbolic tangent)

Similar to the sigmoid (logistic) **function**
(but at least this is zero-centered…)

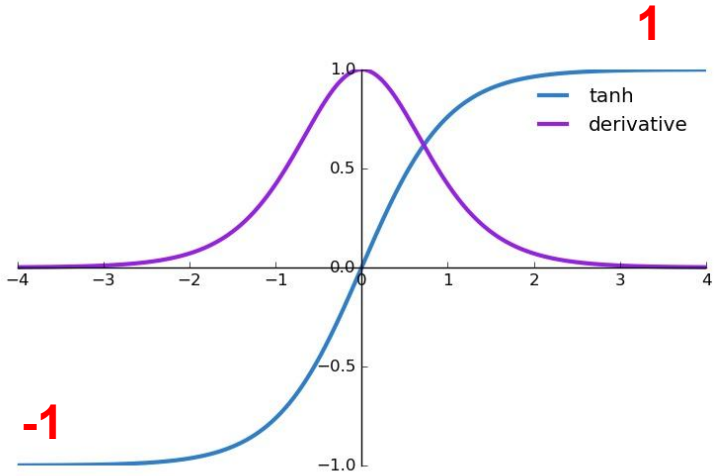


$$\tanh(z) = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\tanh'(z) = 1 - \tanh^2(z)$$

Activation functions

Tanh (Hyperbolic tangent)



Similar to the sigmoid (logistic) **function**
(but at least this is zero-centered…)

Consequence: The use of sigmoid and Tanh activation functions is uncommon inside deep neural networks, with the exception of a few specialized roles…

$$\tanh(z) = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\tanh'(z) = 1 - \tanh^2(z)$$

A better alternative: ReLU

Activation functions

ReLU (Rectified Linear Unit)

If it receives a negative number as input, it sets the gradient to zero.
If it receives a negative input for every element in the training set, it “switches off”.

$$ReLU(z) = max(0, z)$$

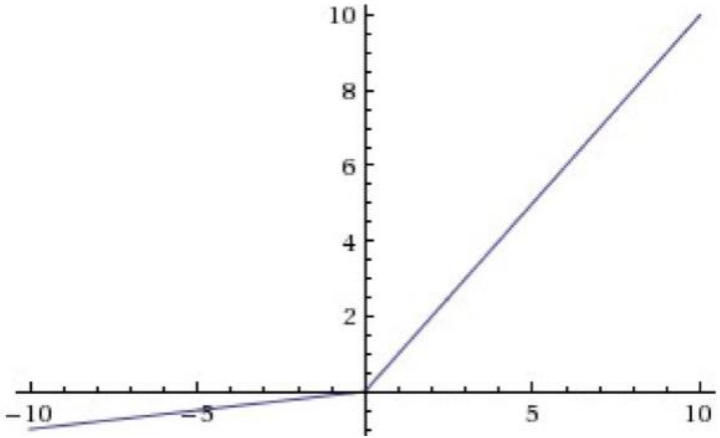
$$\text{if } z < 0 \ : ReLU'(z) = 0$$

$$\text{if } z \geq 0 \ : ReLU'(z) = 1$$

Activation functions

Leaky-ReLU

It solves the "switch-off" problem with ReLU.
In practice, however, ReLU almost always works well.



$$LeakyReLU(z) = max(\beta z, z)$$

$$0 < \beta < 1$$

$$\text{if } z < 0 \ : LeakyReLU'(z) = \beta$$

$$\text{if } z \geq 0 \ : LeakyReLU'(z) = 1$$

The scaling of the gradient

Let's examine **how the layer affects the gradient** back-propagated through it!

$$\hat{y} = g(wx + b) \qquad w, b, x, \hat{y} \in \mathbb{R}$$

$$\frac{\partial J}{\partial x} = \frac{\partial \hat{y}}{\partial x} \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

Besides the activation function **the magnitude of the weights** can also cause problems.

The scaling of the gradient

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

If the absolute value of w and $g'(wx + b)$ is expected to be **less than 1**:

The scaling of the gradient

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

If the absolute value of w and $g'(wx + b)$ is expected to be **less than 1**:

→ By stacking many layers, the back-propagated **gradient becomes the product of many small numbers**, and its expected value **decreases progressively** as it moves toward the front of the network.

→ **“Vanishing gradient” problem**

The scaling of the gradient

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

If the absolute value of w and $g'(wx + b)$ is expected to be **less than 1**:

→ By stacking many layers, the back-propagated **gradient becomes the product of many small numbers**, and its expected value **decreases progressively** as it moves toward the front of the network.

→ **“Vanishing gradient” problem**

Consequence: Near the input of the network, learning will become orders of magnitude slower than at the end of the network.

The scaling of the gradient

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

If the absolute value of w and $g'(wx + b)$ is expected to be **greater than 1**:

The scaling of the gradient

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

If the absolute value of w and $g'(wx + b)$ is expected to be **greater than 1**:

→ As layers are stacked, the back-propagated **gradient becomes the product of many large numbers**, and its expected value **increases** as it moves toward the front of the network.

→ **“Exploding gradient” problem**

The scaling of the gradient

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

If the absolute value of w and $g'(wx + b)$ is expected to be **greater than 1**:

→ As layers are stacked, the back-propagated **gradient becomes the product of many large numbers**, and its expected value **increases** as it moves toward the front of the network.

→ **“Exploding gradient” problem**

Consequence: Near the input of the network, the gradient updates will be orders of magnitude larger than at the end of the network, so the affected parameters do not converge during training.

The unstable gradient problem

In general, the **product of an increasing number of random variables becomes less and less likely to be close to 1**.

→ **Unstable gradient problem**

The unstable gradient problem

In general, the **product of an increasing number of random variables becomes less and less likely to be close to 1**.

→ **Unstable gradient problem**

In a network with too many layers, different parts of the network may learn at completely different speeds: while the updates for some parameters are negligible, others are so large that their learning does not converge.

Addressing the unstable gradient problem

The unstable gradient problem prevents us from training really deep neural networks.

Unfortunately, the network does not learn to scale the back-propagated gradient on its own.

Solution?

Addressing the unstable gradient problem

Batch normalization (2015)

We rescale the gradient every few layers so that its magnitude remains similar across all sections of the network.

How do we rescale the gradient?

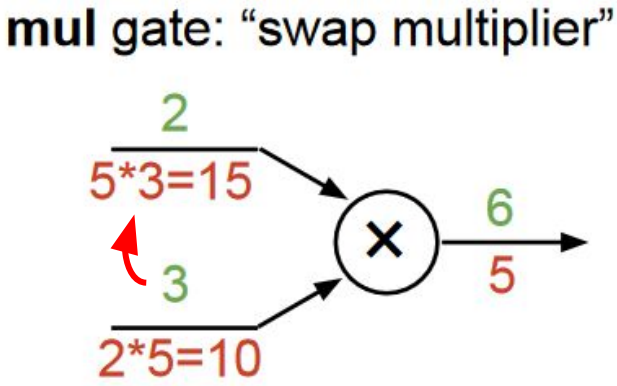
Addressing the unstable gradient problem

Batch normalization (2015)

We rescale the gradient every few layers so that its magnitude remains similar across all sections of the network.

How do we rescale the gradient?

If we multiply by a constant during the forward pass, we multiply the gradient by that number during the backward pass too!



Batch normalization

$$\hat{x}_i := \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

We standardize the variables individually (mean-std normalization) **based on the mean and standard deviation of the last one or more data batches**

$$\hat{y}_i := w \cdot \hat{x}_i + b$$

Also, we learn an additional scaling of the variables (parameters w, b)

$$w, b \in \mathbb{R}$$

Batch normalization

$$\hat{x}_i := \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

The **mean** and **standard deviation** of the x_i variables within the last few batches.

$$\hat{y}_i := w \cdot \hat{x}_i + b$$

After standardization, x^ has a mean of 0 and a standard deviation of 1; **however, we do not necessarily want variables of this magnitude**, so we also apply an additional scaling.

$$w, b \in \mathbb{R}$$

Addressing the unstable gradient problem

The unstable gradient problem prevents us from training really deep neural networks.

Unfortunately, the network does not learn to scale the back-propagated gradient on its own.

Solution:

- Batch normalization

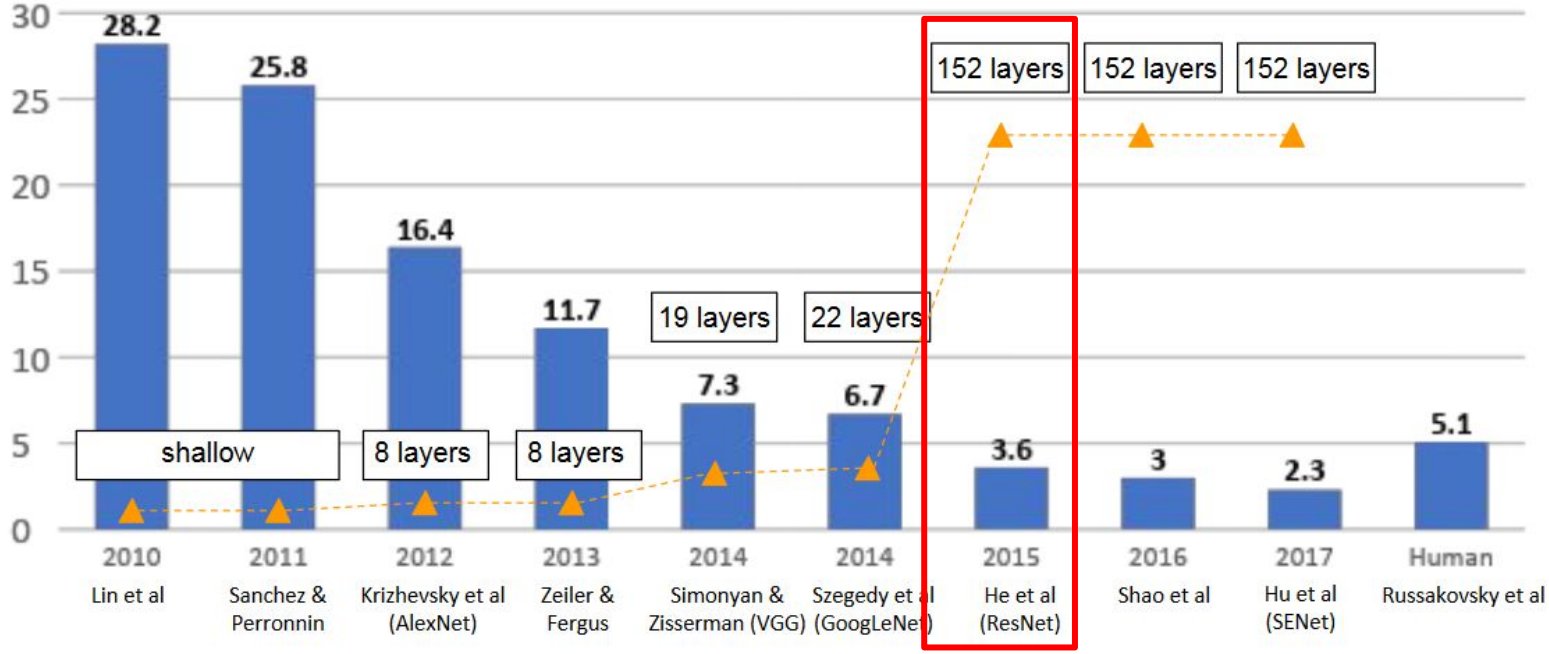
Other solutions?

Image source: Stanford CS231n

Performance on the ImageNet dataset - History

ImageNet ILSVRC Challenge Winners - every year

(1,000-class image classification, 1-class accuracy over 5 trials)



Addressing the unstable gradient problem

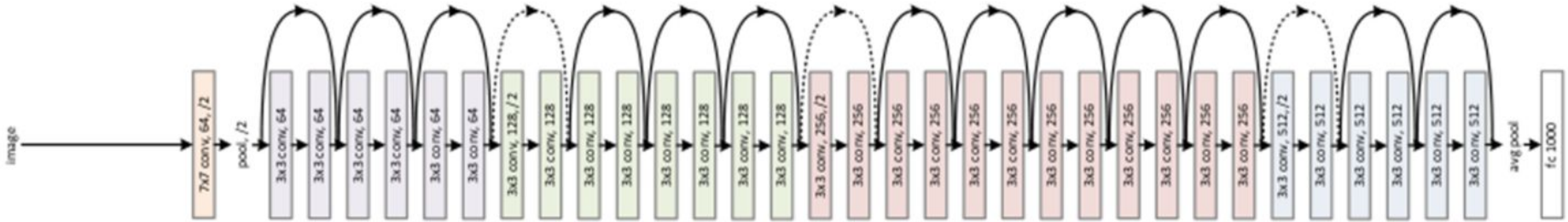
Residual Networks - ResNet (2015)

Let’s maintain a **separate path** in the computation graph along which the gradient can be back-propagated without disruption.

“Gradient highway”

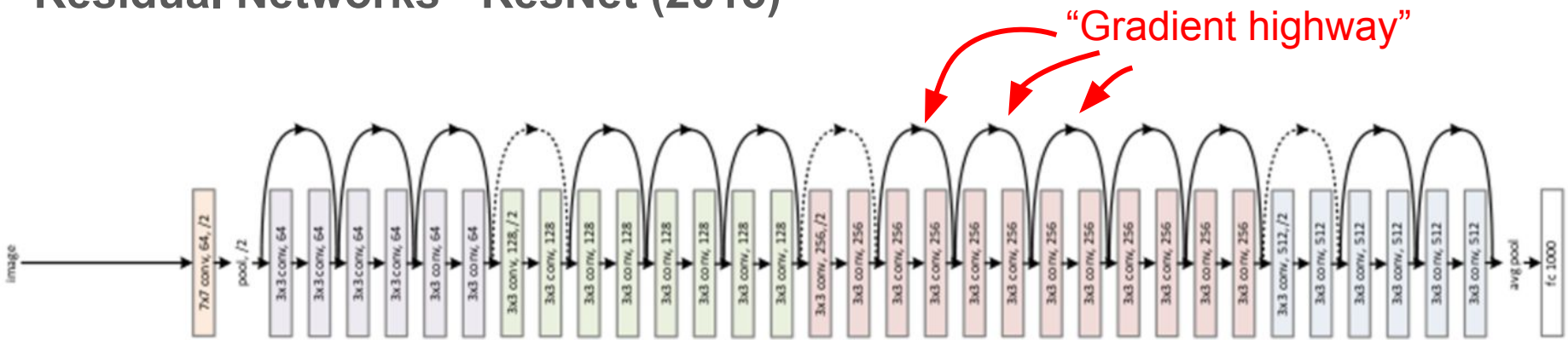
Addressing the unstable gradient problem

Residual Networks - ResNet (2015)



Addressing the unstable gradient problem

Residual Networks - ResNet (2015)



Deep learning - Residual networks

Residual Networks - ResNet (2015)

Superhuman performance!

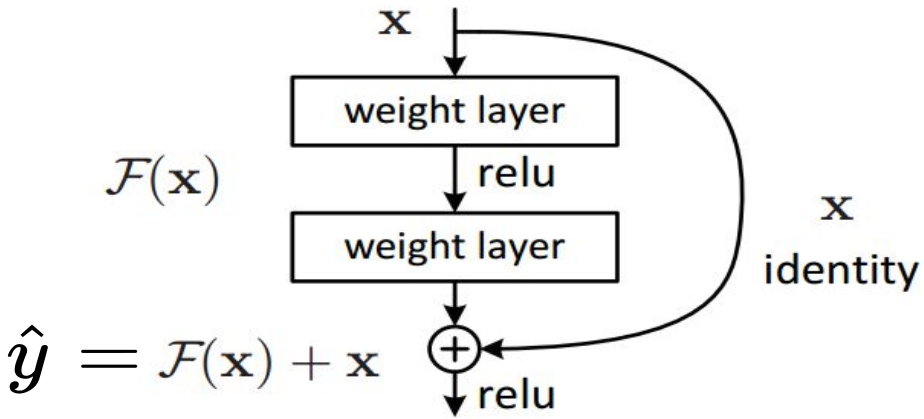
- **ILSVRC:** 96.4% accuracy across 1,000 image categories (top-5)
- **up to 1,500 convolutional layers, max-pooling, 1 fully connected layer**
- **60.1 million parameters*,** of which 90+% are in the convolutional layers
- **Feed-forward:** 4.1 billion floating-point operations* (FLOPs)

*ResNet-151

Addressing the unstable gradient problem

Residual block: We add the input to the output

Why is this beneficial?



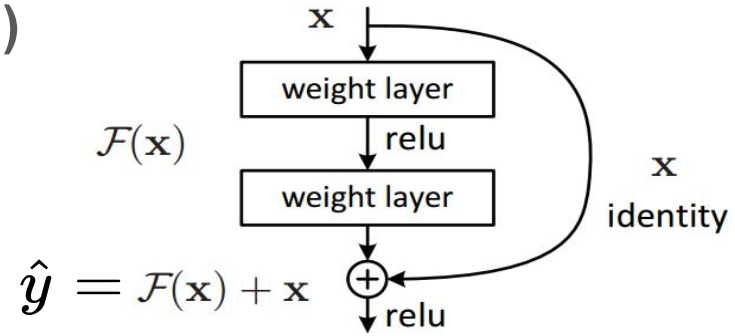
Addressing the unstable gradient problem

The advantages of the residual block (1)

$$\hat{y} = F(x) + x$$

$$\frac{\partial \hat{y}}{\partial x} = \ ?$$

$$\frac{\partial J}{\partial x} = \ ?$$



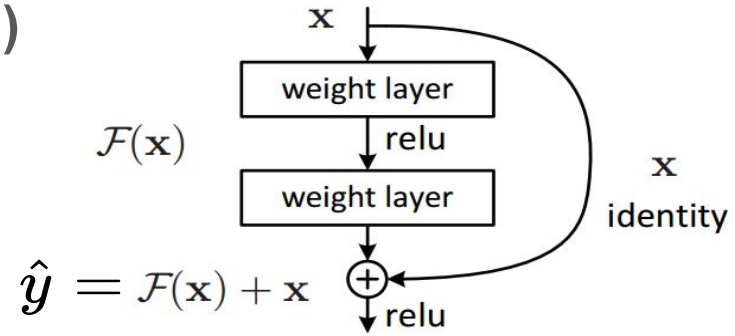
Addressing the unstable gradient problem

The advantages of the residual block (1)

$$\hat{y} = F(x) + x$$

$$\frac{\partial \hat{y}}{\partial x} = F'(x) + 1$$

$$\frac{\partial J}{\partial x} = (F'(x) + 1) \cdot \frac{\partial J}{\partial \hat{y}} = F'(x)\frac{\partial J}{\partial \hat{y}} + \frac{\partial J}{\partial \hat{y}}$$



Addressing the unstable gradient problem

The advantages of the residual block (1)

$$\hat{y} = F(x) + x$$

$$\frac{\partial \hat{y}}{\partial x} = F'(x) + 1$$

$$\frac{\partial J}{\partial x} = (F'(x) + 1) \cdot \frac{\partial J}{\partial \hat{y}} = F'(x)\frac{\partial J}{\partial \hat{y}} + \boxed{\frac{\partial J}{\partial \hat{y}}}$$

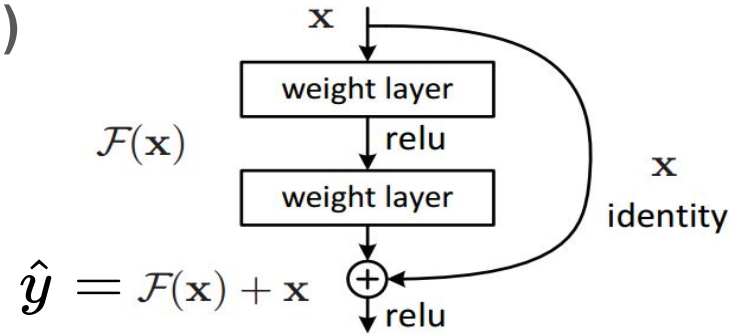
The gradient coming from the subsequent layer can continue toward the front of the network without disruption. If the first term is small enough, the magnitude of the gradient remains unchanged...

(For convenience: x, y are scalars here)

Addressing the unstable gradient problem

The advantages of the residual block (2)

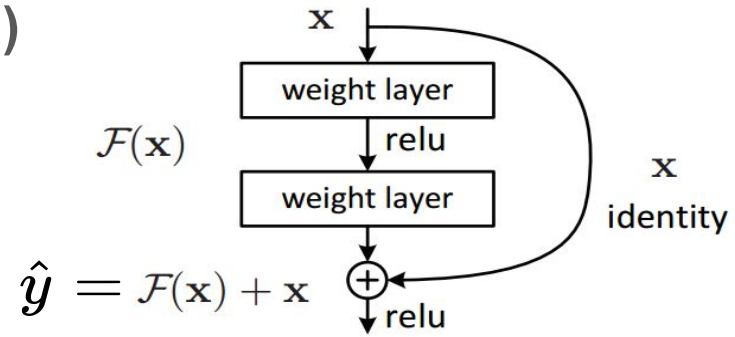
$$\hat{y} = F(x) + x$$



Addressing the unstable gradient problem

The advantages of the residual block (2)

$$\hat{y} = F(x) + x$$



An alternative interpretation: With residual networks, it is much easier to implement a series of small processing steps. With weights initialized to zero, the residual block acts as an identity operation, simply returning the input. By stacking many such layers or blocks - each modified only slightly - it can learn to form complex processing pipelines from a series of small processing steps.
In contrast, the identity operation is not easy to produce with traditional layers containing activation functions.

Deep neural networks today

MobileNet v2 (2018)

```
mobilenet_model = torch.hub.load('pytorch/vision:v0.10.0', 'mobilenet_v2', pretrained=True)
mobilenet_model.eval();

print(mobilenet_model)
```

```
Downloading: "https://github.com/pytorch/vision/zipball/v0.10.0" to /root/.cache/torch/hub/v0.10.0.zi
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to /root/.cache/torch/hu
100%|██████████| 13.6M/13.6M [00:00<00:00, 69.3MB/s]MobileNetV2(
  (features): Sequential(
    (0): ConvBNActivation(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU6(inplace=True)
    )
    (1): InvertedResidual(
      (conv): Sequential(
        (0): ConvBNActivation(
          (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=Fals
          (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

Deep neural networks today

MobileNet v2 (2018)

```
mobilenet_model = torch.hub.load('pytorch/vision:v0.10.0', 'mobilenet_v2', pretrained=True)
mobilenet_model.eval();

print(mobilenet_model)
```

```
Downloading: "https://github.com/pytorch/vision/zipball/v0.10.0" to /root/.cache/torch/hub/v0.10.0.zi
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to /root/.cache/torch/hu
100%|██████████| 13.6M/13.6M [00:00<00:00, 69.3MB/s]MobileNetV2(
  (features): Sequential(
    (0): ConvBNActivation(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU6(inplace=True)
    )
    (1): InvertedResidual(
      (conv): Sequential(
        (0): ConvBNActivation(
          (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=Fals
          (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

BatchNorm

Residual block

Residual block

Addressing the unstable gradient problem

Ensuring **proper scaling of the gradient** is one of the most important challenges in deep learning.

Image source: Simonyan & Zisserman 2014

VGG-19

- 92.7% top-5 accuracy (ILSVRC)
- 19 layers with parameters
- 143 million parameters
- One feed-forward: 20 GFLOPS
- Originally, **it took 2–3 weeks to train on 4 high-end GPUs.**

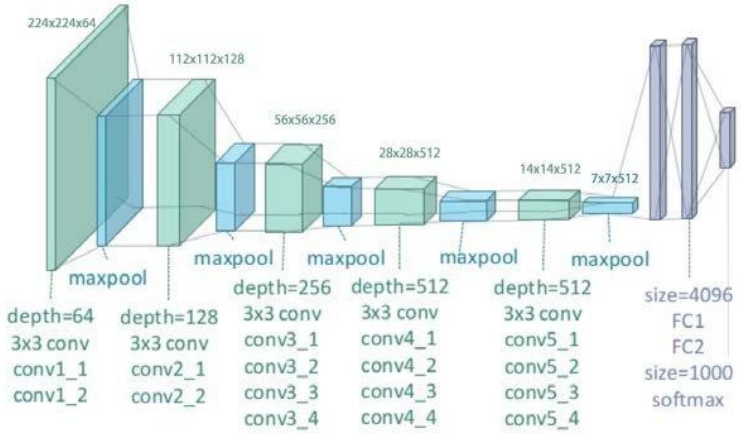
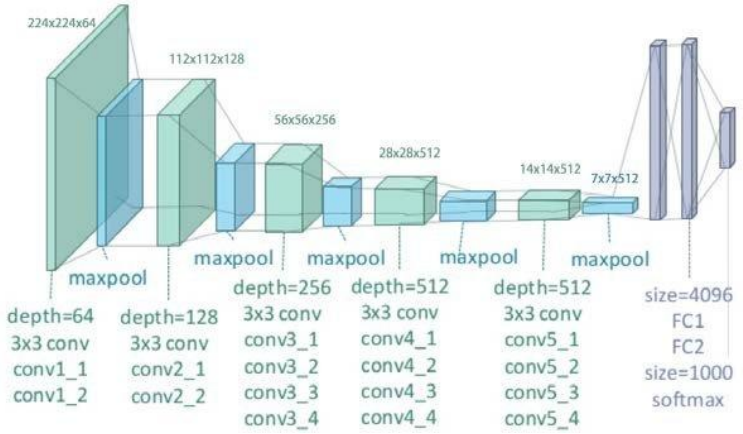


Image source: Simonyan & Zisserman 2014

VGG-19

- 92.7% top-5 accuracy (ILSVRC)
- 19 layers with parameters
- 143 million parameters
- One feed-forward: 20 GFLOPS
- Originally, **it took 2–3 weeks to train on 4 high-end GPUs.**



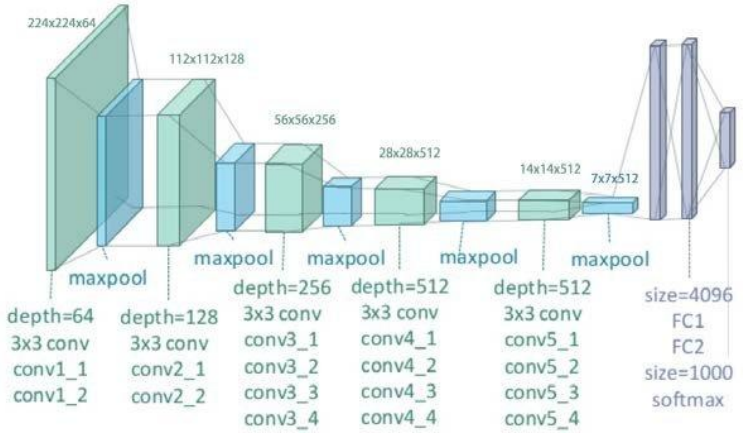
Training deep neural networks from scratch requires significant computing power.

Does that mean it's pointless to attempt to train them for meaningful tasks without high-end hardware?

Image source: Simonyan & Zisserman 2014

VGG-19

- 92.7% top-5 accuracy (ILSVRC)
- 19 layers with parameters
- 143 million parameters
- One feed-forward: 20 GFLOPS
- Originally, **it took 2–3 weeks to train on 4 high-end GPUs.**



Training deep neural networks from scratch requires significant computing power.

Does that mean it's pointless to attempt to train them for meaningful tasks without high-end hardware?

Fortunately, no!

Transfer learning

We need to recognize that **it is not necessary to start training from scratch for every single task.**

Transfer learning: The process of utilizing knowledge gained while learning one type of task to solve another task.

Transfer learning - example

Example:

Classifying dogs and cats in high-resolution photos.
We have 500 examples of each for training.

How do we solve this?

- Let's train a **really deep** convolutional neural network!

Result:

- It overfits quickly. It's not really possible to train millions of parameters well with just a few hundred images, even with data augmentation. **:(**

Transfer learning - example

Example:

Classifying dogs and cats in high-resolution photos.
We have 500 examples of each for training.

How do we solve this?

- Let’s train a **smaller** convolutional network!

Result:

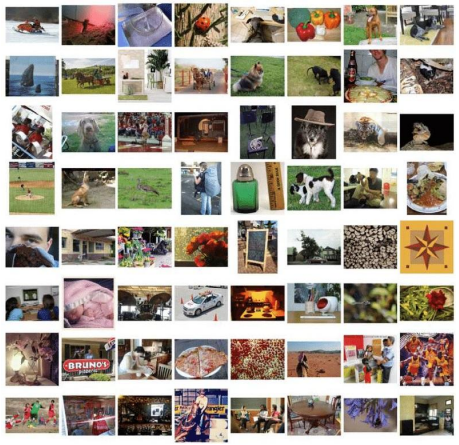
- Poor performance. A network with few parameters won’t be able to handle complex tasks well. **:(**

Transfer learning - example

Instead:

1. Let’s find a much larger dataset that is somewhat similar to our target dataset!

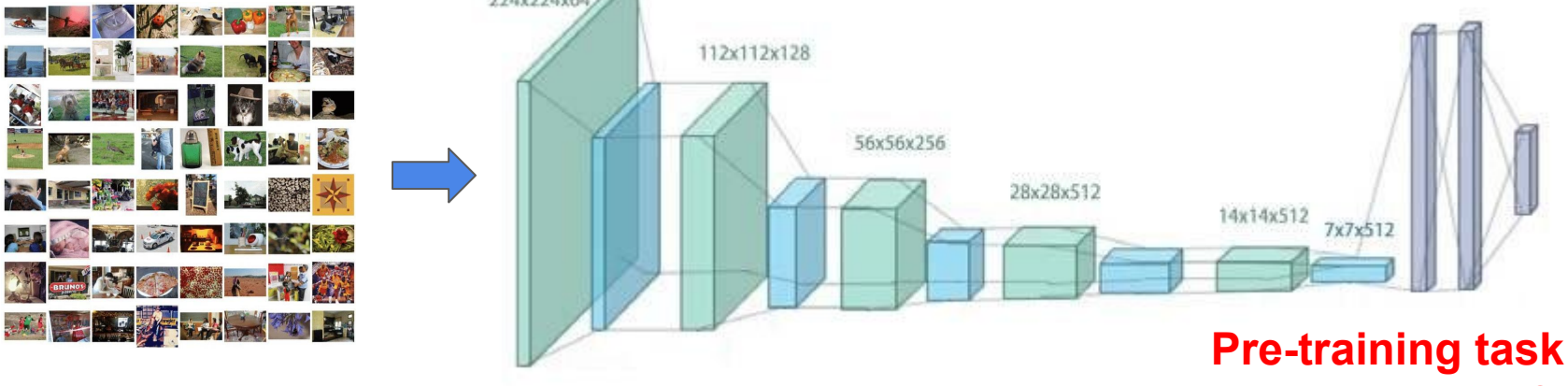
IMAGENET



Transfer learning - example

Instead:

2. Let's train a really deep neural network on the large dataset!

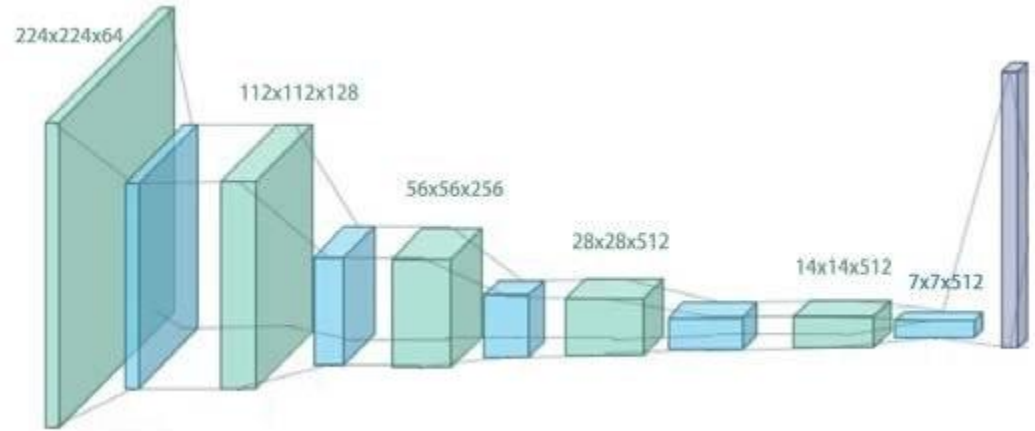


Pre-training task:
1000-way classification

Transfer learning - example

Instead:

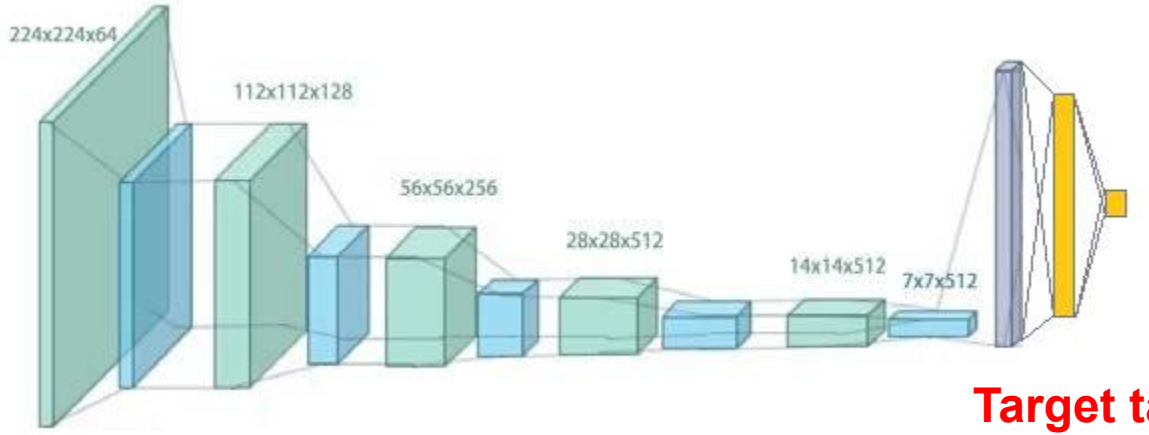
3. Discard the final layers of the **pretrained network**.



Transfer learning - example

Instead:

4. Add new layers to the truncated pretrained network that are suitable in shape and function for the target task.

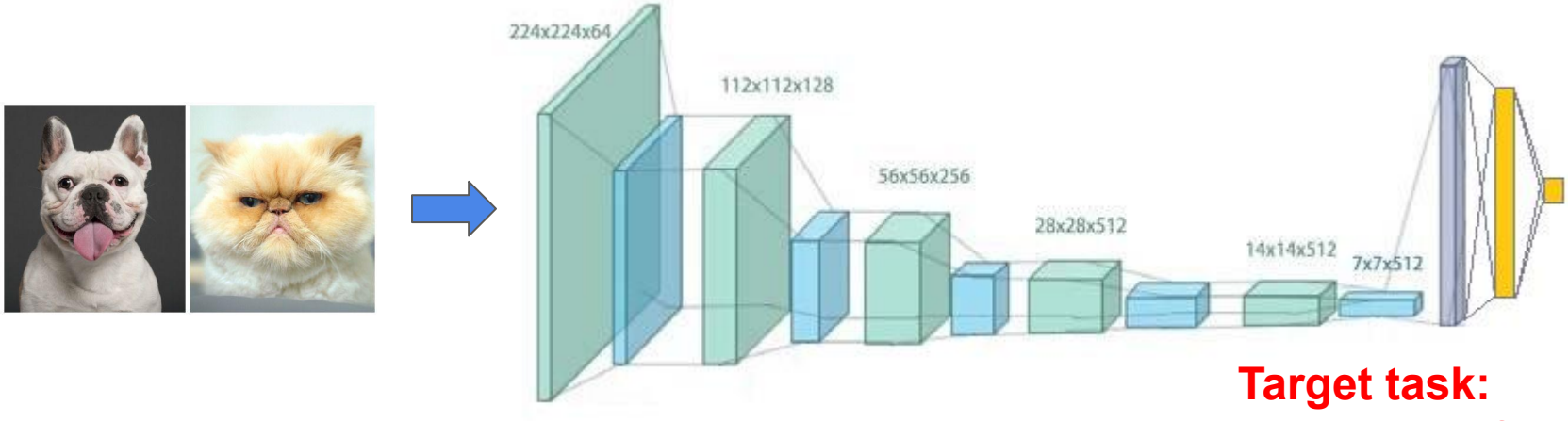


Target task:
Binary classification

Transfer learning - example

Instead:

5. **Fine-tune the network** on the target task.



Target task:
Binary classification

Transfer learning - example

Transfer learning (pretraining - fine-tuning)**:**

1. Let's find a much larger dataset that is somewhat similar to our target dataset!
2. Let's train a really deep neural network on the large dataset!
3. Discard the final layers of the **pretrained network**.
4. Add new layers to the truncated pretrained network that are suitable in shape and function for the target task.
5. **Fine-tune the network** on the target task.

Transfer learning - example

Millions of trainable parameters →
Large dataset is required
(100k+ samples)**, long training.**

Transfer learning (pretraining - fine-tuning)**:**

1. Let's find a much larger dataset that is somewhat similar to our target dataset!
2. Let's train a really deep neural network on the large dataset!
3. Discard the final layers of the **pretrained network**.
4. Add new layers to the truncated pretrained network that are suitable in shape and function for the target task.
5. **Fine-tune the network** on the target task.

Only a few thousand parameters to train, a small database is sufficient
(a few hundred or thousand samples)**, quick training**

Transfer learning - example

~~**Millions of trainable parameters →**~~
~~**Large dataset is required**~~
~~(100k+ samples)~~**~~, long training.~~**

Transfer learning (pretraining - fine-tuning)**:**

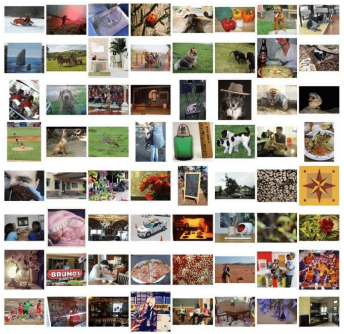
1. **Let's find a pretrained deep neural network** based on a task or dataset closely related to our target task!
2. Discard the final layers of the **pretrained network**.
3. Add new layers to the truncated pretrained network that are suitable in shape and function for the target task.
4. **Fine-tune the network** on the target task.

Only a few thousand parameters to train, a small database is sufficient
(a few hundred or thousand samples)**, quick training**

Transfer learning - example

It’s not always easy to find an appropriate pre-training database or a pre-trained model for every task…

Pretraining task

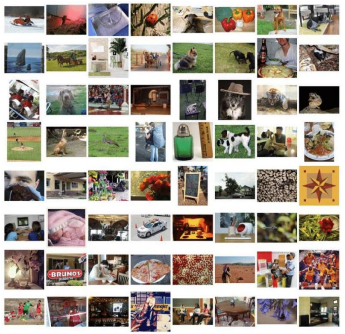


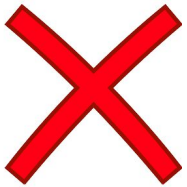
Target task

Transfer learning - example

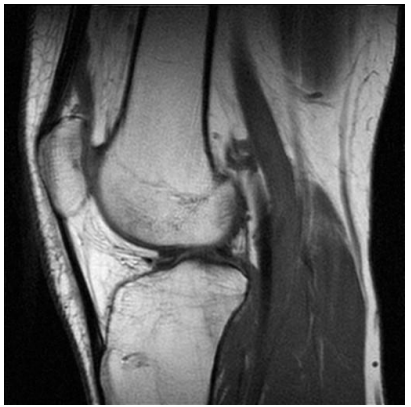
It’s not always easy to find an appropriate pre-training database or a pre-trained model for every task…

Pretraining task





Target task



Transfer learning - Weight freezing

We can freeze all or some of the original parameters of the pretrained model before fine-tuning it (weight freezing)**.**

- This can reduce the extent of overfitting on the target task, as it reduces the number of trainable parameters…

Transfer learning - Weight freezing

We can freeze all or some of the original parameters of the pretrained model before fine-tuning it (weight freezing)**.**

- This can reduce the extent of overfitting on the target task, as it reduces the number of trainable parameters…

For example:

```
for param in model.parameters():
    param.requires_grad = False
```

Transfer learning - Weight freezing

Transfer Learning in Practice - General “rules”

- **The smaller the dataset for the target task:**
  The fewer layers/parameters we train during fine-tuning.

Transfer learning - Weight freezing

Transfer Learning in Practice - General “rules”

- **The smaller the dataset for the target task:**
  The fewer layers/parameters we train during fine-tuning.

- **The more the target task differs from the pretraining task:**
  The fewer pre-trained layers we should retain for fine-tuning.