Deep Learning for Vision (DLV)

Classification - part I

Denis Coquenet

2024-2025

Université de Rennes

istic Informatique Électronique

Goals of this course

Knowledge

- What is classification
- Evolution of deep architectures for classification
- Key components for stable training of deep models

Skills and know-how

- Formalize classification as a deep learning task
- Evaluate a classification model
- Compare neuronal layers in terms of parameters, output shape, memory consumption, computational cost and input constraints

Table of contents

1. The image classification task
   - What is image classification?
   - Why is it useful?
   - How to handle this task?
   - Problem formulation
   - Evaluating the task
   - Defining a training objective
2. Case study: LeNet for digit recognition
3. Towards deep neural networks

The image classification task

What is the main subject in the image?

Input: image

Output: Butterfly

Output: Candle

Output: Car

Task: find the main object in the image

Fine-grained classification

The expected class is domain-specific, *e.g.*, cat species:

Input: image

Output: Angora

Output: Bengal

Output: Persan

The image classification task

Why?

- Autonomous cars
- Tagging images (keyword)
- Security: facial recognition
- Knowledge: mushroom identification (https://shroom.id/)

How?

With deep supervised learning!

Constraints

- Classes must be known beforehand
- Requires annotated data for each class
- Item can be anywhere in the image (position, size)

Supervised learning formulation

Goal

Given a set of c classes, we want to learn a function $f_\theta : \mathcal{X} \to \mathcal{Y}$ which associates a class to each image.

Need: annotated data

$\mathcal{D}_{\text{train}} = \{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}\}_{i=1}^{n}$: a set of n training images

$x_i \in \mathbb{R}^{H_i \times W_i \times C_i}$: an image of height H_i, width W_i and encoded on C_i channels

($C_i = 1$ for gray-scaled, $C_i = 3$ for RGB)

$y_i \in \{0, 1\}^{\mathsf{N}_c}$ the one-hot encoded class

Example for three classes: butterfly, candle and car

Butterfly $\to [1, 0, 0]$
Candle $\to [0, 1, 0]$
Car $\to [0, 0, 1]$

Softmax

Let o be the output of the network: $o = f_\theta(x)$
Class probabilities can be obtained through softmax activation:

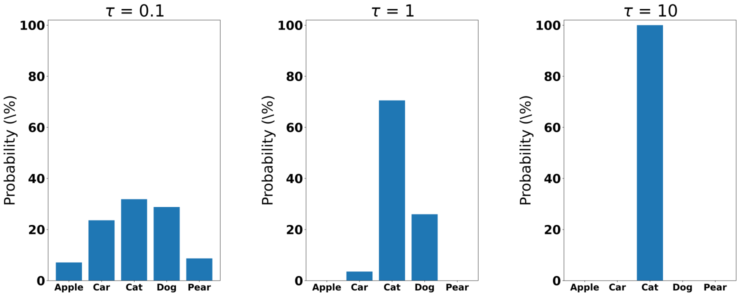
$$\hat{y} = \text{softmax}(o) \quad \Leftrightarrow \quad \hat{y}_i = \frac{e^{o_i}}{\sum_{j=1}^{c} e^{o_j}}$$

Example for three classes: butterfly, candle and car

Class	o_i	$\hat{y}_i$
Butterfly	11.5	21.42%
Candle	12.8	78.58%
Car	-2.4	0.00%

Adding a temperature factor τ enables to modulate the sharpness of the probability distribution: $\hat{y}_i = \dfrac{e^{o_i/\tau}}{\displaystyle\sum_{j=1}^{c} e^{o_j/\tau}}$

- Could be optimized at training time through gradient descent
- Could be used at inference time to bring stochasticity (NLP)



Metrics

Top-1 accuracy:

$$a(\hat{y}, y) = \begin{cases} 1 & \text{if } \arg\max(\hat{y}) = \arg\max(y) \\ 0 & \text{otherwise} \end{cases}$$

Similarly, top-5 accuracy.

Goal

$$\min_\theta \frac{1}{n} \sum_{i=1}^{n} a(f_\theta(x_i), y_i)$$

But a not differentiable.

Cross-entropy loss

$$\begin{aligned}\mathcal{L}_{\mathsf{CE}}(\hat{y}, y) &= -\textstyle\sum_{j=1}^{c} y_j \log(\hat{y}_j) \\ &= -y_{c^*} \log(\hat{y}_{c^*})\end{aligned}$$

where c^* is the index of the ground truth class.

New goal

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{\mathsf{CE}}(f_\theta(x_i), y_i)$$

Now the cost function is differentiable, but...

Training minimizes the error over the training set only

Metric computations

Compute the top-1 accuracy and top-5 accuracy for the following predictions/ground truth

# Sample	Scores							Ground truth
	Apple	Bike	Car	Cat	Dog	Pear	Plane	
1	15	10	2	5	20	8	1	Apple
2	2	5	10	8	4	12	3	Car
3	12	2	1	6	4	5	9	Car
4	1	2	3	4	5	6	7	Plane
5	7	6	5	4	3	2	1	Bike

# Sample	Scores Apple	Bike	Car	Cat	Dog	Pear	Plane	Ground truth	Top-1 acc.	Top-5 acc.
1	15	10	3	5	**20**	8	2	Apple	✗	✓
2	2	5	10	8	4	**12**	3	Car	✗	✓
3	**12**	2	1	6	4	5	9	Car	✗	✗
4	1	2	3	4	5	6	**7**	Plane	✓	✓
5	**7**	6	5	4	3	2	1	Bike	✗	✓
									20%	80%

Remark: the softmax function is monotonic

➤ the argmax can be used directly on scores at prediction time

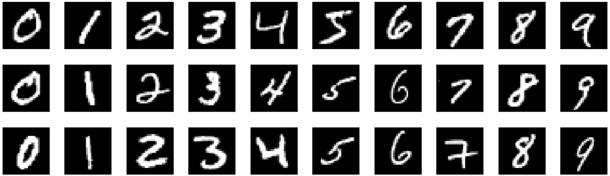
1 The image classification task

2 Case study: LeNet for digit recognition

- The MNIST dataset
- LeNet-5 architecture
- Pytorch implementation of the whole task
- Training and evaluation

3 Towards deep neural networks

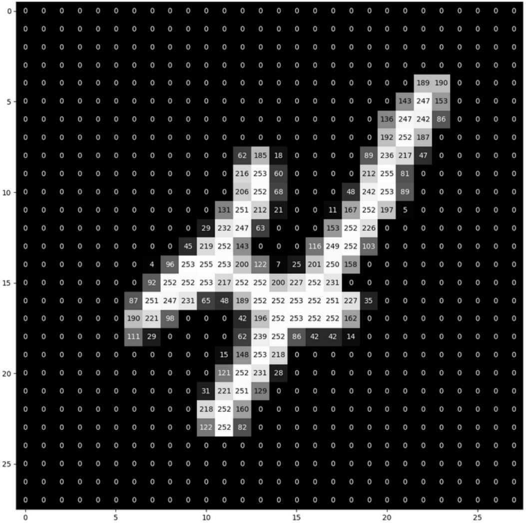
The MNIST dataset (1998) [1]



A handwritten digit classification dataset

- Gray-scaled images of size 28×28
- 10 classes: digits from 0 to 9
- 60,000 training samples + 10,000 test samples

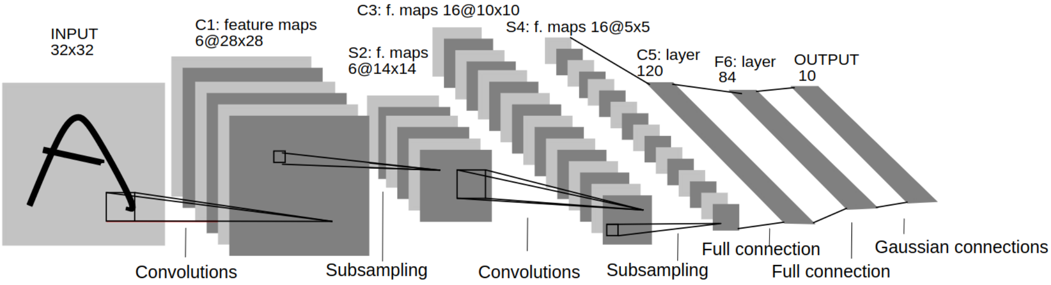
The MNIST dataset (1998) [1]



Very low resolution

Gray-scaled: only one dimension for color ($28 \times 28 \times 1 \rightarrow 784$ values in total)

LeNet-5 architecture (1998) [1]



A Convolutional Neural Network (CNN)

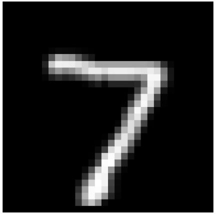
- 2 convolutions followed by max pooling
- 3 fully-connected layers
- Originally designed for inputs of size 32×32

Pytorch implementation: architecture

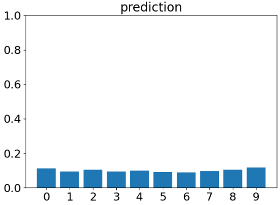
```python
from torch import nn
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0)
        self.fc1 = nn.Linear(400, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        # Non-parametric
        self.max_pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        out = torch.tanh(self.conv1(x))
        out = self.max_pool(out)
        out = torch.tanh(self.conv2(out))
        out = self.max_pool(out)
        out = out.reshape(out.size(0), -1)
        out = torch.tanh(self.fc1(out))
        out = torch.tanh(self.fc2(out))
        out = self.fc3(out)
        return out
```

```python
# Model instanciation
net = LeNet()

# Summary
from torchsummary import summary
summary(net, (1, 32, 32))

# Forward pass
# x: input image (1, 1, 32, 32)
o = net(x) # (1, 10)
hat_y = torch.softmax(o, dim=1)
```

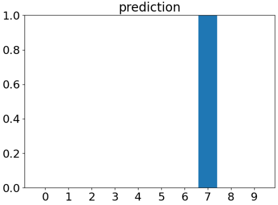
```
=================================================================
Layer (type:depth-idx)       Output Shape            Param #
=================================================================
Conv2d: 1-1                  [-1, 6, 28, 28]          156
MaxPool2d: 1-2               [-1, 6, 14, 14]          --
Conv2d: 1-3                  [-1, 16, 10, 10]         2,416
MaxPool2d: 1-4               [-1, 16, 5, 5]           --
Linear: 1-5                  [-1, 120]                48,120
Linear: 1-6                  [-1, 84]                 10,164
Linear: 1-7                  [-1, 10]                 850
==================================================================
Total params: 61,706
==================================================================
```



Input image x



Before training



After training

Pytorch implementation: training

```python
from torch.utils.data import DataLoader
from torchvision.transforms import Compose, ToTensor, Resize
from torchvision.datasets import MNIST

num_epochs = 10
batch_size = 100
learning_rate = 0.01
transform = Compose([ToTensor(), Resize((32, 32))])
train_loader = DataLoader(MNIST(root="./cache", train=True, download=True, transform=transform),
        batch_size=batch_size, shuffle=True)
test_loader = DataLoader(MNIST(root="./cache", train=False, download=True, transform=transform),
        batch_size=batch_size, shuffle=False)

device = ("cuda" if torch.cuda.is_available() else "cpu")
net = LeNet().to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
loss_fn = torch.nn.CrossEntropyLoss()

for epoch in range(num_epochs):
    train_epoch(train_loader, net, optimizer, loss_fn)
    eval(test_loader, net)
```

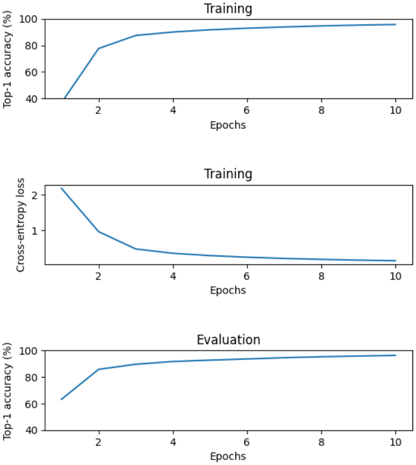
Pytorch implementation: training

```python
def train_epoch(dataloader, net, optimizer, loss_fn):
    epoch_loss = list()
    epoch_top1_acc = list()
    net.train()
    for x, y in dataloader:
        batch_loss, batch_top1_acc = train_batch(x, y, net, optimizer, loss_fn)
        epoch_loss.append(batch_loss)
        epoch_top1_acc.append(batch_top1_acc)
        current_loss = np.mean(epoch_loss)
        current_top1_acc = 100 * np.mean(epoch_top1_acc)
    print(f"Train epoch {epoch+1}: loss: {current_loss:.4f} ; top-1 accuracy: {current_top1_acc:.2f}%")

def train_batch(x, y, net, optimizer, loss_function):
    x, y = x.to(device), y.to(device)
    optimizer.zero_grad()  # zero the gradient buffers
    output = net(x)
    loss = loss_function(output, y)
    loss.backward()
    optimizer.step()
    top1_acc = compute_top1_acc(output, y)
    return loss.item(), top1_acc.item()
```

```python
def compute_top1_acc(prediction, ground_truth):
    # prediction (B, N), ground_truth (B)
    best_prediction = torch.argmax(prediction, dim=1)
    return torch.mean(best_prediction == ground_truth, dtype=torch.float)

def eval_batch(x, y, net):
    x, y = x.to(device), y.to(device)
    output = net(x)
    top1_acc = compute_top1_acc(output, y)
    return top1_acc.item()

def eval(dataloader, net):
    top1_acc = list()
    net.eval()
    with torch.no_grad():
        for x, y in dataloader:
            batch_top1_acc = eval_batch(x, y, net)
            top1_acc.append(batch_top1_acc)
    top1_acc = 100 * np.mean(top1_acc)
    print(f"Eval epoch {epoch+1}: top-1 accuracy: {top1_acc:.2f}%")
```

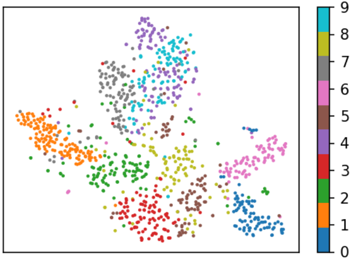
Let's run it!

```
Train epoch 1: loss: 2.1790 ; top-1 accuracy: 34.73%
Eval epoch 1: top-1 accuracy: 57.64%
Train epoch 2: loss: 1.0242 ; top-1 accuracy: 73.68%
Eval epoch 2: top-1 accuracy: 84.61%
Train epoch 3: loss: 0.5052 ; top-1 accuracy: 86.74%
Eval epoch 3: top-1 accuracy: 88.76%
Train epoch 4: loss: 0.3701 ; top-1 accuracy: 89.77%
Eval epoch 4: top-1 accuracy: 91.27%
Train epoch 5: loss: 0.2959 ; top-1 accuracy: 91.62%
Eval epoch 5: top-1 accuracy: 92.71%
Train epoch 6: loss: 0.2447 ; top-1 accuracy: 93.05%
Eval epoch 6: top-1 accuracy: 93.82%
Train epoch 7: loss: 0.2068 ; top-1 accuracy: 94.17%
Eval epoch 7: top-1 accuracy: 94.70%
Train epoch 8: loss: 0.1780 ; top-1 accuracy: 95.00%
Eval epoch 8: top-1 accuracy: 95.44%
Train epoch 9: loss: 0.1555 ; top-1 accuracy: 95.62%
Eval epoch 9: top-1 accuracy: 96.04%
Train epoch 10: loss: 0.1384 ; top-1 accuracy: 96.04%
Eval epoch 10: top-1 accuracy: 96.50%
```

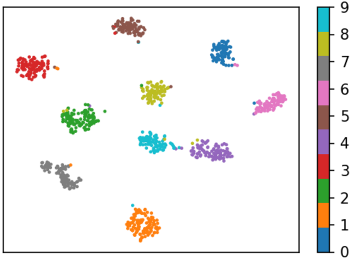


T-SNE [2]

T-SNE (T-distributed Stochastic Neighbor Embedding):
Approach to reduce dimensions by preserving relative distance between points from input space to output space.



T-SNE on raw data



T-SNE on latent representation

Table of contents

1 The image classification task

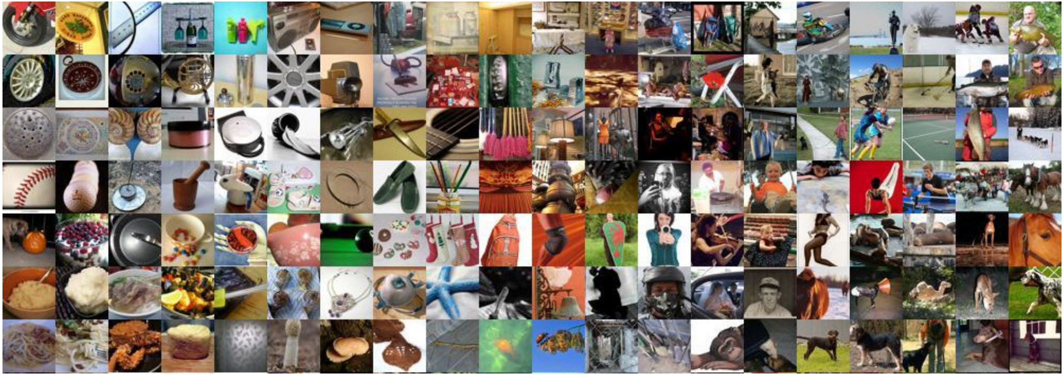
2 Case study: LeNet for digit recognition

3 Towards deep neural networks
- The ImageNet dataset
- AlexNet
- VGG
- GoogleNet
- Depthwise Separable Convolution
- ResNet

The ImageNet dataset (2012) [3]

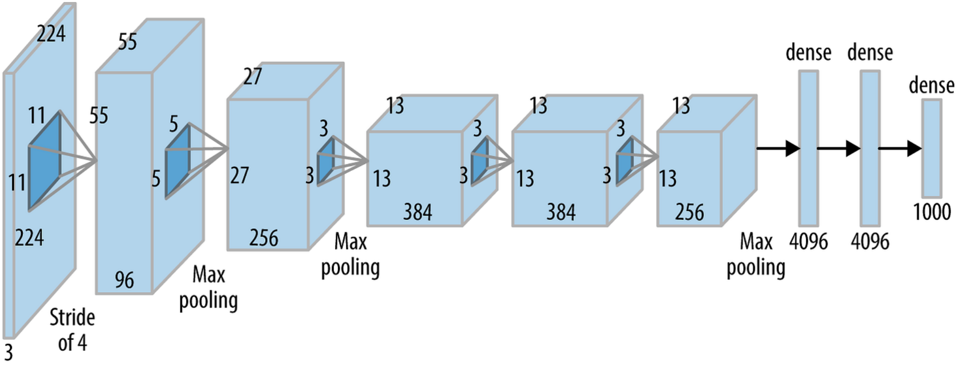
A large-scale dataset for image classification

- 1,000 classes
- 1.2 M training images (average size: 469x387 pixels)



source: https://cs.stanford.edu/people/karpathy/cnnembed/

AlexNet (2012) [3]



source: oreilly.com

- 8-layer model (5 convolutions + 3 denses)
- ∼60 M parameters
- Top-5 accuracy on ImageNet: 83.6%

Model definition

```python
from torch import nn

class AlexNet(nn.Module):
    def __init__(self, num_classes=10):
        super(AlexNet, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 96, kernel_size=11, stride=4, padding=2),
            nn.BatchNorm2d(96),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2))
        self.conv2 = nn.Sequential(
            nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2))
        self.conv3 = nn.Sequential(
            nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU())
        self.conv4 = nn.Sequential(
            nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU())
        self.conv5 = nn.Sequential(
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2))
        self.fc1 = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(9216, 4096),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(4096, num_classes))

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.conv5(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.fc3(out)
        return out
```

Model analysis

```python
# Instanciation
net = AlexNet()

# Summary
from torchsummary import summary
summary(net, (3, 224, 224))
```

```
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 96, 55, 55]          34,944
       BatchNorm2d-2           [-1, 96, 55, 55]             192
              ReLU-3           [-1, 96, 55, 55]               0
         MaxPool2d-4           [-1, 96, 27, 27]               0
            Conv2d-5          [-1, 256, 27, 27]         614,656
       BatchNorm2d-6          [-1, 256, 27, 27]             512
              ReLU-7          [-1, 256, 27, 27]               0
         MaxPool2d-8          [-1, 256, 13, 13]               0
            Conv2d-9          [-1, 384, 13, 13]         885,120
      BatchNorm2d-10          [-1, 384, 13, 13]             768
             ReLU-11          [-1, 384, 13, 13]               0
           Conv2d-12          [-1, 384, 13, 13]       1,327,488
      BatchNorm2d-13          [-1, 384, 13, 13]             768
             ReLU-14          [-1, 384, 13, 13]               0
           Conv2d-15          [-1, 256, 13, 13]         884,992
      BatchNorm2d-16          [-1, 256, 13, 13]             512
             ReLU-17          [-1, 256, 13, 13]               0
        MaxPool2d-18            [-1, 256, 6, 6]               0
          Dropout-19                 [-1, 9216]               0
           Linear-20                 [-1, 4096]      37,752,832
             ReLU-21                 [-1, 4096]               0
          Dropout-22                 [-1, 4096]               0
           Linear-23                 [-1, 4096]      16,781,312
             ReLU-24                 [-1, 4096]               0
           Linear-25                   [-1, 10]          40,970
================================================================
Total params: 58,325,066
Input size (MB): 0.57
Params size (MB): 222.49
```

Feature extraction

- Convolution for local feature extraction (shift-equivariance)
- Dense for global feature extraction

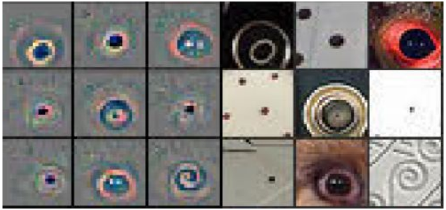
Regularization

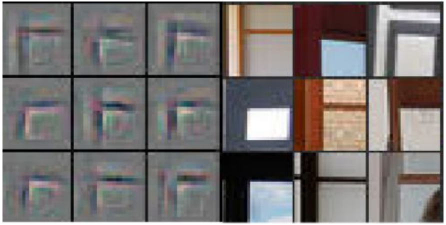
- BatchNorm
- Dropout
- Data augmentation

Activation

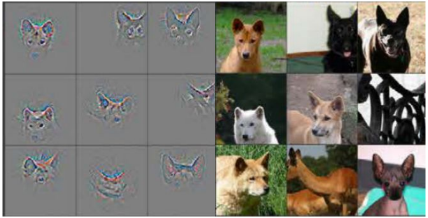
- ReLU activation for vanishing gradient issue

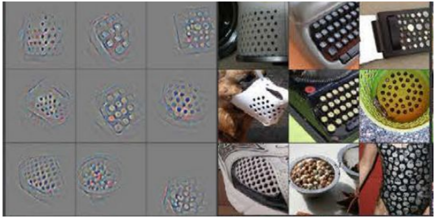
Layer 2

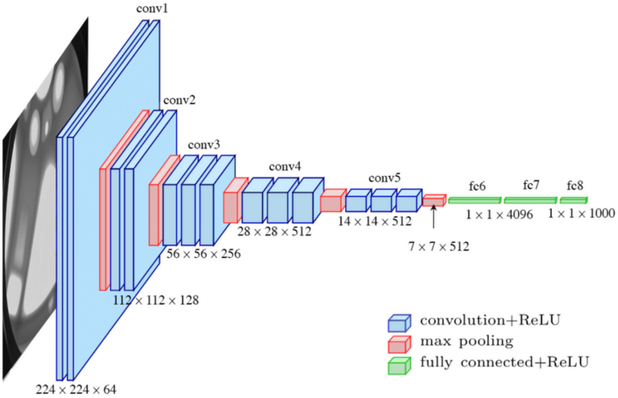




Layer 5







VGG-16

source: Ferguson *et al.*, International Conference on Big Data, 2017

VGG: a family of models

	VGG-11	VGG-13	VGG-16	VGG-19	VGG-19 [384]
# conv. layers # dense layers # parameters (M)	8 3 133	10 3 133	13 3 138	16 3 144	16 3 144
ImageNet (valid) Top-1 accuracy (%)	70.4	71.3	73.0	72.7	**73.1**

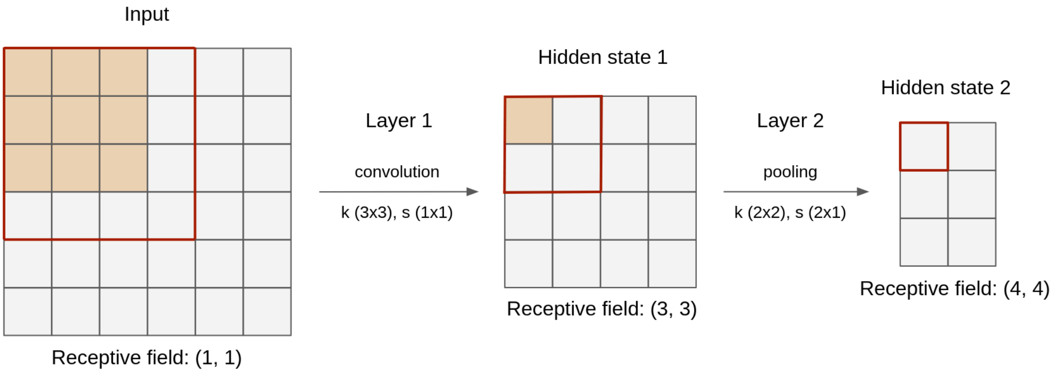
VGG-19 top-5 accuracy on ImageNet (test): 92.7%

➤ The deeper (and the wider) the better !

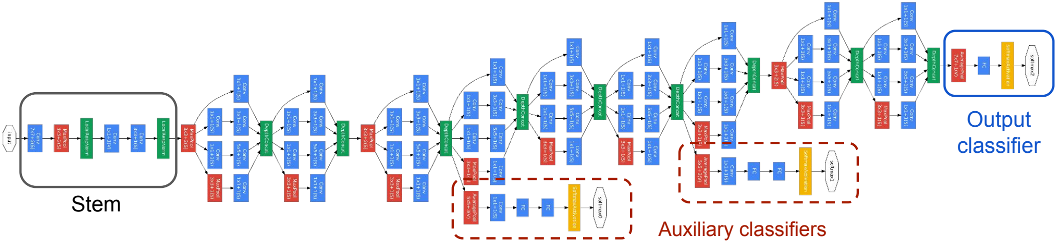
But: # parameters $\nearrow$, memory consumption $\nearrow$, computations $\nearrow$.

Definition

The receptive field corresponds to the size of the region in the input that produces the feature of a given hidden state.



➤ What is the receptive field in the decision layer?



- Conv. stem + stack of "Inception" blocks + output classifier
- Auxiliary classifiers used during training only
- 6.8 M parameters
- Top-5 accuracy on ImageNet: 93.3%

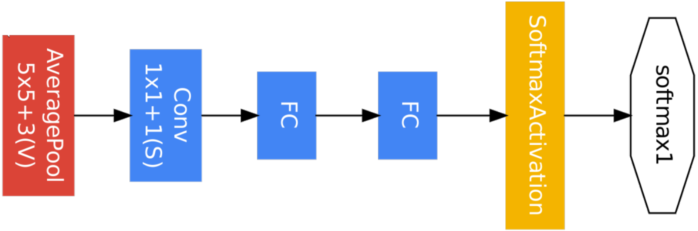
Auxiliary classifiers

Idea

Assumption: latent representations are discrimative enough in the middle of the network.
Goal: improve gradient propagation for lower layers.

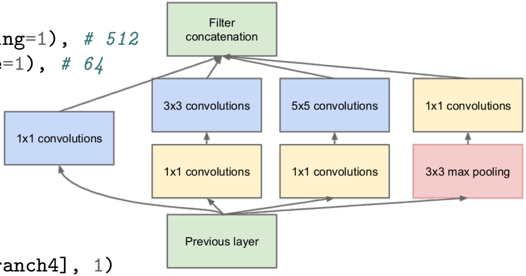
Global loss

$$\mathcal{L} = \mathcal{L}_{\text{output}} + 0.3\mathcal{L}_{\text{aux}_1} + 0.3\mathcal{L}_{\text{aux}_2}$$



Inception block

```python
class Inception(nn.Module):
    def __init__(self, in_channels, out_br1, red_br2, out_br2, red_br3, out_br3, out_br4):
        super().__init__()
        self.branch1 = nn.Conv2d(in_channels, out_br1, kernel_size=1) # 128
        self.branch2 = nn.Sequential(
            nn.Conv2d(in_channels, red_br2, kernel_size=1), # 128
            nn.Conv2d(red_br2, out_br2, kernel_size=3, padding=1) # 256
        )
        self.branch3 = nn.Sequential(
            nn.Conv2d(in_channels, red_br3, kernel_size=1), # 24
            nn.Conv2d(red_br3, out_br3, kernel_size=5, padding=1), # 64
        )
        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1), # 512
            nn.Conv2d(in_channels, out_br4, kernel_size=1), # 64
        )

    def forward(self, x): # (B, 512, H, W)
        branch1 = self.branch1(x) # (B, 128, H, W)
        branch2 = self.branch2(x) # (B, 256, H, W)
        branch3 = self.branch3(x) # (B, 64, H, W)
        branch4 = self.branch4(x) # (B, 64, H, W)
        return torch.cat([branch1, branch2, branch3, branch4], 1)
```

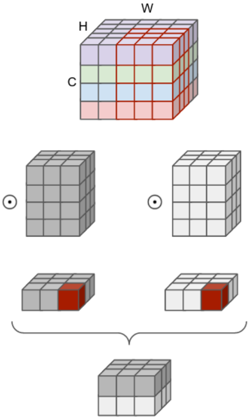


Inception block

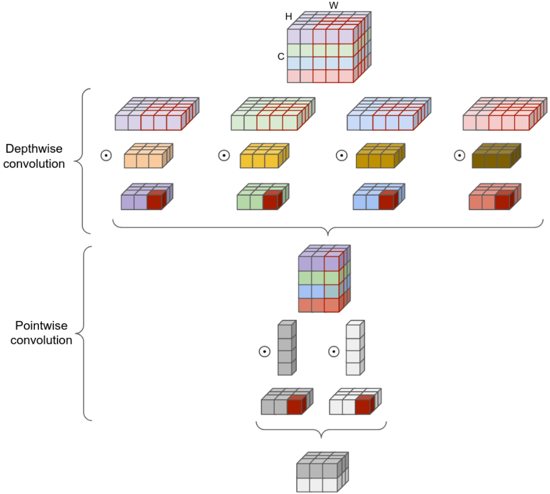
```python
# x: latent representation of dimension (1, 512, H, W)

block = Inception(in_channels=512, out_br1=128, red_br2=128,
                  out_br2=256, red_br3=24, out_br3=64, out_br4=64)
out = block(x) # (1, 512, H, W) using 510,104 parameters

conv = nn.Conv2d(512, 512, kernel_size=3)
out = conv(x) # (1, 512, H, W) using 2,359,808 parameters
```

➤ 4 times less parameters

Another way to reduce the number of parameters: Depthwise Separable Convolutions



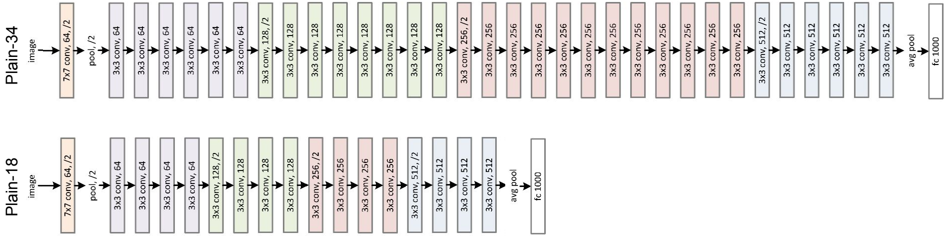
Standard convolution:
k kernels $3 \times 3 \times C$

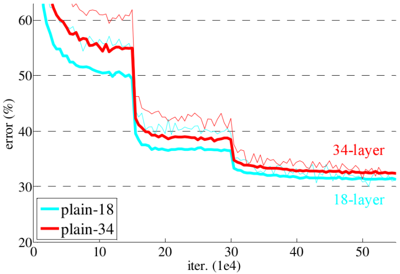


Depthwise separable convolution:
C kernels $3 \times 3 \times 1$ followed by k kernels $1 \times 1 \times C$

Depthwise Separable Convolutions (DSC) [7]

C	n_k	k_H	k_W	# weights convolution $C \times k_\mathrm{H} \times k_\mathrm{W} \times n_\mathrm{k}$	# weights DSC $C \times (k_\mathrm{H} \times k_\mathrm{W} + n_\mathrm{k})$
512	512	3	3	2.4M	0.3M
1028	1028	3	3	9.5M	1.1M
1028	1028	5	5	26.4M	1.1M

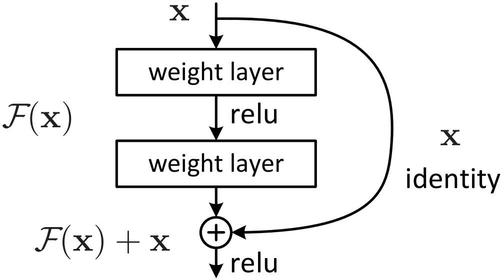




- Stacking more layers leads to poorer results
- Not an overfiting issue (same behaviour during training)
- Why? If shallower network was optimal, additional layers should tend to identity mapping through training.

Goal: to make optimization easier

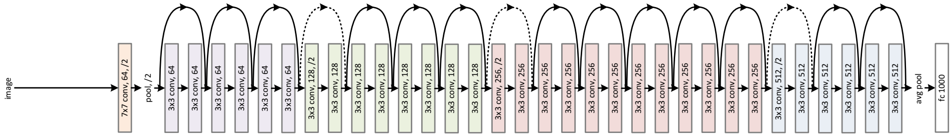
```
class Residual(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += identity
        out = self.relu(out)
        return out
```



Assumption: it is easier to learn the residual mapping $\mathcal{F}(x) = \mathcal{H}(x) - x$ than directly $\mathcal{H}$.
Easier to set weights to 0 (in case identity mapping is optimal) than to learn the identity mapping through non-linear functions.

➤ Improve gradient propagation + multi-scale feature extraction

ResNet (2016) [8]

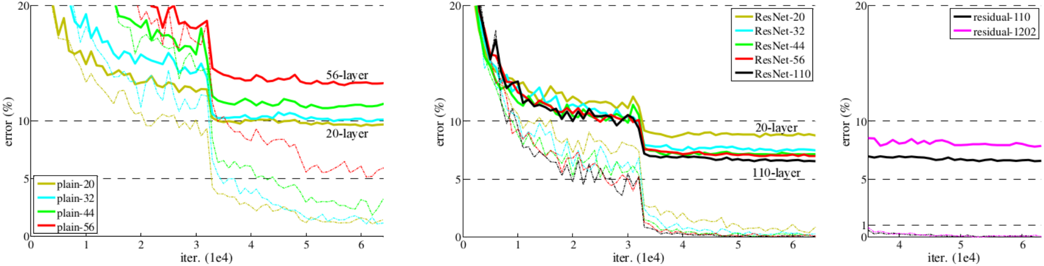


layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Top-5 accuracy: 96.43% for the deepest model

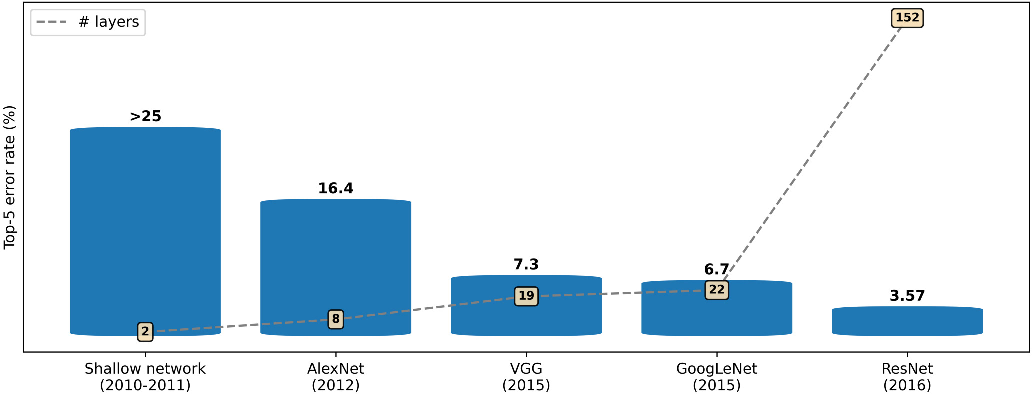
CIFAR 10

- 50,000 training images (32×32)
- 10 classes



Training curves (dashed lines) and test curves (bold lines)

The deeper the better ?!



Classification performance on ImageNet

We talked a lot about architectures, but...

Architecture is not everything, be careful with benchmarks!

- Training time, number of epochs/iterations, mini-batch size
- Weight initialization, optimizer, initial learning rate, learning rate scheduler
- Dropout, normalization, activation functions
- Pre-processing, post-processing

What is really due to architecture novelty?
What is due to training strategy?

➤ Really difficult to fairly compare approaches

Question

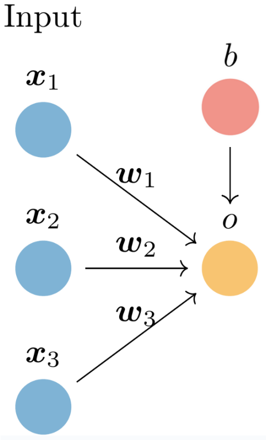
Input image of size 28×28, floats encoded on 4 bytes

Compute the number of FLoating point OPerations (FLOPs), the number of parameters, the output shape and the output tensor memory occupation when applying the following layers independently:

- Fully-connected layers made up of 256 neurons
- Convolutional layer with 256 kernels of size 3×3, stride 1×1 and no padding
- Max Pooling with kernel size 2×2, stride 2×2 and no padding

Biases are considered in the computations

Example



Perceptron example = fully-connected with single output

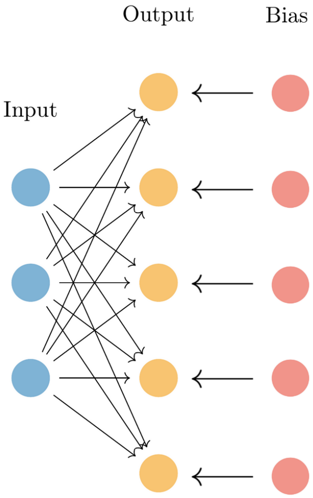
Input shape: 3 (vector)
Output shape: 1 (scalar)
Output size: $1 \times 4 = 4$ bytes (memory occupation)

Number of parameters: 4 $(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3, b)$

Two kinds of operation:

- Multiply-Accumulate Computations (MAC):
  Dot product: $\hat{o} = \boldsymbol{w}_1\boldsymbol{x}_1 + \boldsymbol{w}_2\boldsymbol{x}_2 + \boldsymbol{w}_3\boldsymbol{x}_3$
  3 MAC = 6 FLOPs (1 MAC = 2 FLOPs)
- Addition:
  Bias: $o = \hat{o} + b$ (1 FLOP)

➤ Total: 7 FLOPs



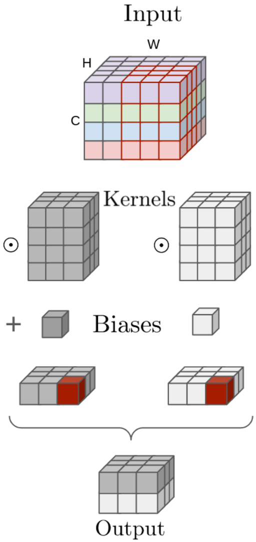
Fully-connected layers made up of 256 neurons

Input image shape: $28 \times 28 = 784$
Output shape: 256
Output size: $256 \times 4 = 1,024$ bytes

Number of parameters:
weights (#output $\times$ #input) + biases (#output)
$256 \times 784 + 256 = 200,960$ parameters

Number of FLOPs:
$2 \times$ #output $\times$ #input $+$ #biases
$2 \times 256 \times 784 + 256 = 401,664$ FLOPs

Correction



Convolutional layer with 256 kernels of size 3×3, stride 1×1 and no padding

W: width, H: height, L: length
K: kernel size, P: padding, S: stride

Input image shape: $28 \times 28 \times 1$
Output shape:
$W_o = H_o = \frac{L_i - K + P}{S} + 1 = \frac{28-3+0}{1} + 1 = 26 \rightarrow 26 \times 26 \times 256$
Output size: $26 \times 26 \times 256 \times 4 = 692,224$ bytes

Number of parameters:
weights (#kernel $\times$ kSize) + biases (#kernel)
$256 \times (3 \times 3 \times 1) + 256 = 2,560$ parameters

Number of FLOPs:
#kernel $\times$ #opsPerKernel $\times$ $(2 \times$ kSize + #biasPerKernel$)$
$256 \times (26 \times 26) \times (2 \times 3 \times 3 \times 1 + 1) = 3,288,064$ FLOPs

0.2	-0.1	**0.1**	-0.2
0.3	**0.7**	-0.8	0
0.1	**0.6**	**0.9**	0.2
0.5	0.1	-0.2	0
1.2	0.3	**0.6**	0.3
0.1	0.4	0.1	0

0.7	0.1	1.2
0.6	0.9	0.6

Max Pooling with kernel size 2×2, stride 2×2 and no padding

W: width, H: height, L: length
K: kernel size, P: padding, S: stride

Input image shape: $28 \times 28 \times 1$
Output shape:
$W_o = H_o = \frac{L_i - K + P}{S} + 1 = \frac{28-2+0}{2} + 1 = 14 \rightarrow 14 \times 14 \times 1$
Output size: $14 \times 14 \times 1 \times 4 = 784$ bytes

Number of parameters: 0
➤ Non-parametric operation, nothing is learned!

Number of FLOPs:
$H_i \times W_i \times C_i = 28 \times 28 \times 1 = 784$ FLOPs
➤ Only select max value, just need to read all values

Quizz

Which layer is parametric?

- ReLU
- Convolution
- Pooling
- Batch Normalization
- Dropout
- Fully-connected
- Softmax

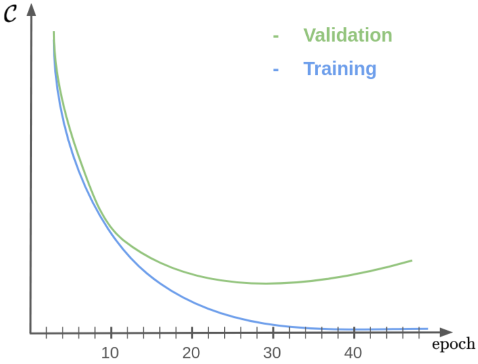
Quizz

Is a fully connected layer...

- Shift-equivariant?
- Shift-invariant?
- Dependant of the input size?
- Adapted to model global context?

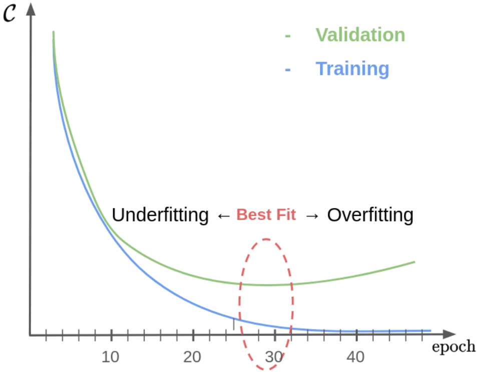
➤ Same question for a convolutional layer

Here are the curves for my cost function during my training on the training and validation sets.



Questions:

1. Which phenomenon can we observe?
2. The weights from which epoch should I keep to use my network for inference?
3. What could cause such phenomenon?
4. What could be done to improve this situation?



Answers:

1. This is a case of overfitting
2. Weights from epoch 29 led to the best results
3. Too many parameters with respect to the number of training data
   ➤ The model learns non-discriminative information (noise) from training data
4. Use more data, dropout, normalization...

Conclusion

Convolutions

- Weights shared through sliding window
- Must stack some of them to enlarge receptive field
- Shift-equivariant property

Architectures

Deeper and deeper: more efficient but...

- requires more data
- requires regularization techniques to stabilize training
- requires more computational resources

➤ Importance of multi-scale information with residual connections

➤ Next time: transformer architecture and how to deal with the lack of data!

References I

[1] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings IEEE* 86.11 (1998), pp. 2278–2324.

[2] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems*. 2012, pp. 1106–1114.

[4] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *13th European Conference on Computer Vision*. Ed. by David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars. Vol. 8689. Lecture Notes in Computer Science. 2014, pp. 818–833.

[5] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*. 2015.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions". In: *Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[7] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: *Conference on Computer Vision and Pattern Recognition*. 2017, pp. 1800–1807.

References II

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.