Introduction to Machine Learning CS182

Lu Sun

School of Information Science and Technology

ShanghaiTech University

December 19, 2023

Today:

- Introduction to deep learning
- Convolutional neural networks (CNN)

Readings:

- Deep Learning (DL), Chapter 9

Today's Agenda

Introduction to Deep Learning

Convolutional Neural Network (CNN)

- Overview
- Architecture
- Optimization and Regularization
- Practice in Python
- Examples and Applications

Introduction to Deep Learning

Introduction

- Our goal: Build intelligent algorithms to make sense of data
  - Example: Recognizing objects in images

→ **red panda** (*Ailurus fulgens*)

  - Example: Predicting what would happen next

Vondrick et al. CVPR2016

Introduction

- A broad range of real-world applications
  - Speech recognition
    - Input: sound wave → Output: transcript
  - Language translation
    - Input: text in language A (Eng) → Output: text in language B (Chs)
  - Image classification
    - Input: images → Output: image category (cat, dog, car, house, etc.)
  - Autonomous driving
    - Input: sensory inputs → Output: actions (straight, left, right, stop, etc.)
- Main challenges: difficult to manually design the algorithms

Introduction

- Each task as a mapping function (or a model)

Input data → Mapping function → Expected output

  - input data: images
  - expected output: object or action names

- Building such mapping functions from data

red panda (*Ailurus fulgens*)

→ Mapping function

Introduction

- Building a mapping function (model)

$$y = f(x; \theta)$$

  - x: input data
  - y: expected output
  - θ: parameters to be estimated

- Learning the model from data
  - Given a dataset $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^{N}$
  - Find the 'best' parameter $\hat{\theta}$, such that

$$y_n \simeq f(x_n; \hat{\theta}) \quad \forall n$$

  - And it can be generalized to unseen input data

What Is Deep Learning?

- Using deep neural networks as the mapping function

- Deep neural networks
  - A family of parametric models
  - Consisting of many 'simple' computational units
  - Constructing a multi-layer representation of input

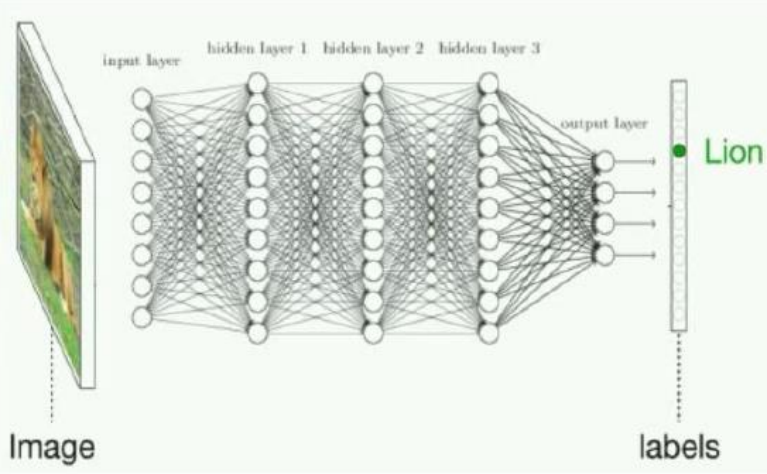
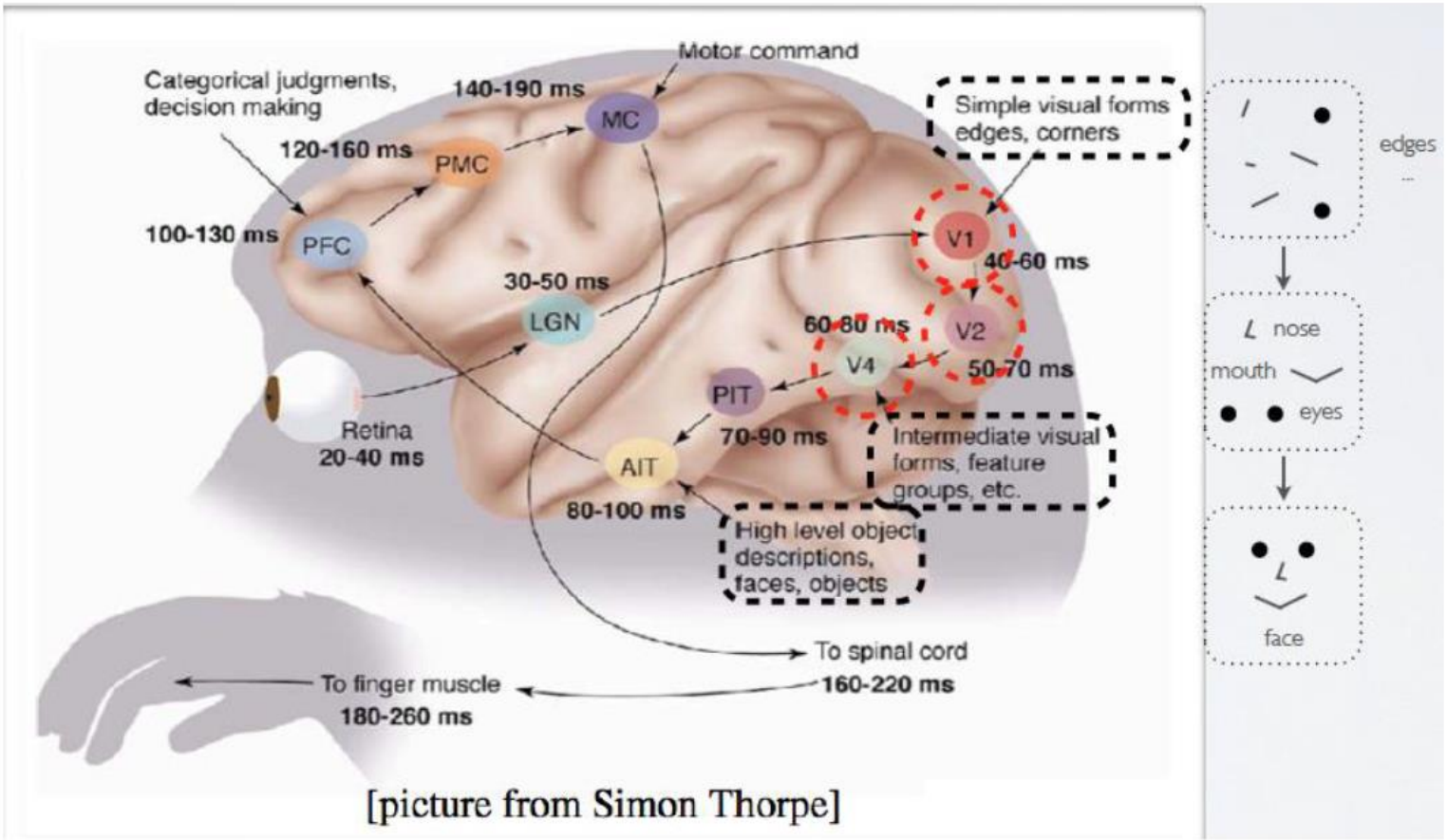


Image from Jeff Clune's Deep Learning Overview

Why Deep Networks?

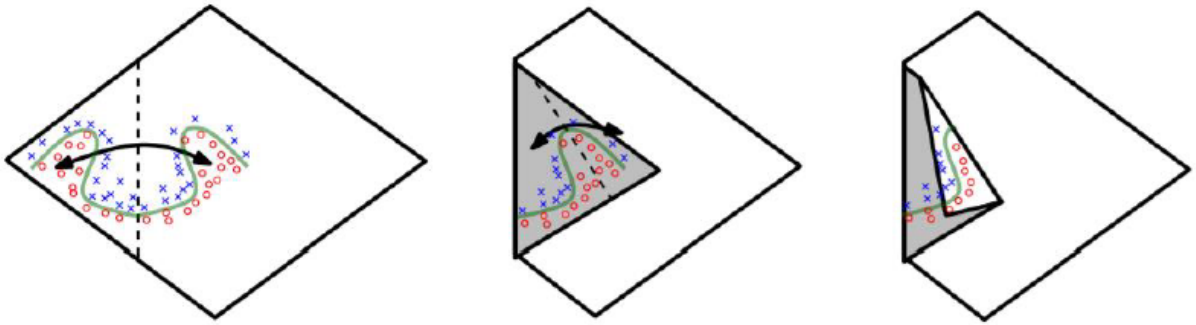
- Inspiration from visual cortex



[picture from Simon Thorpe]

Why Deep Networks?

- A deep architecture can represent certain functions more compactly
  - (Montufar et al., NIPS'14)
    - Functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow one.



Why Deep Networks?

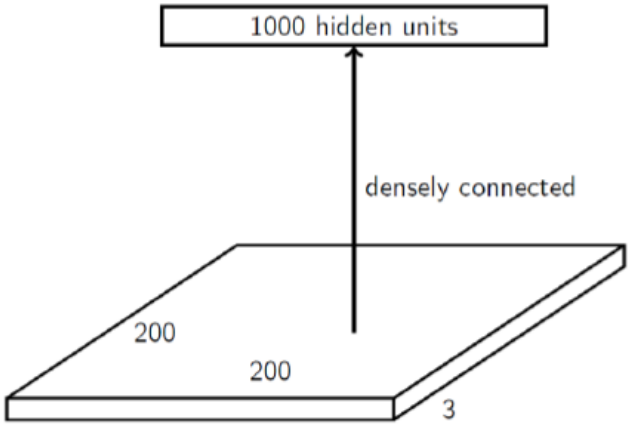
- One of the major thrust areas recently in various pattern recognition, prediction and data analysis
  - Efficient representation of data and computation
  - Other key factors: large datasets and hardware
- The state of the art in many problems
  - Often exceeding previous benchmarks by large margins
  - Achieve better performances than human for certain "complex" tasks.
- But also somewhat controversial …
  - Lack of theoretical understanding
  - Sometimes difficult to make it work in practice

Convolutional Neural Networks-CNN

- Overview
- Architecture
- Optimization and Regularization
- Practice in Python
- Applications

Motivation

- Visual recognition
  - Suppose we aim to train a network that takes a 200x200 RGB image as input



  - What is the problem with have full connections?
    - Too many parameters! 200x200x3x1000 = 120 million
    - What happens if the object in the image shifts a little?

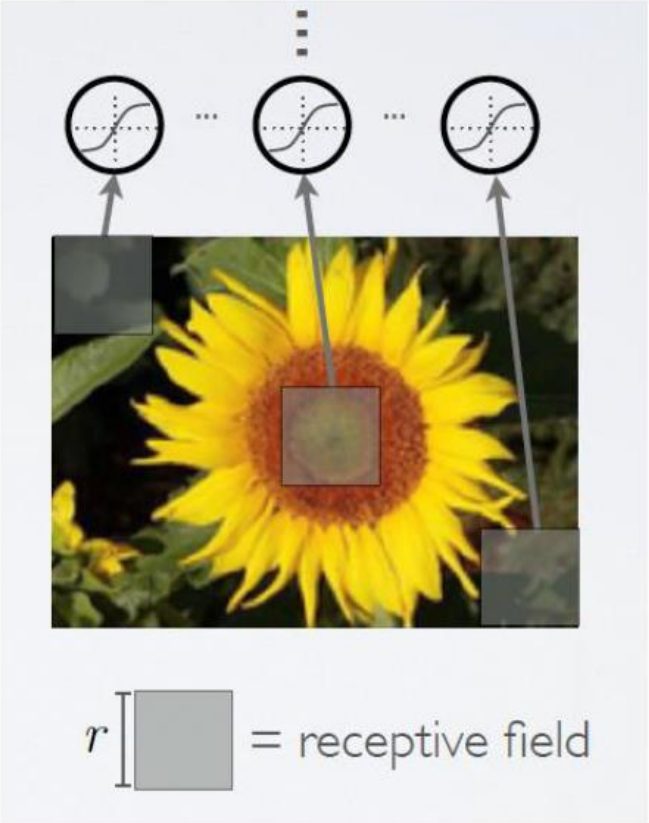
Motivation

- Visual Recognition: Design a neural network that
  - Much deal with very high-dimensional inputs
  - Can exploit the 2D topology of pixels in images
  - Can build in invariance to certain variations we can expect
    - Translation, small deformations, illumination, etc.

red panda (*Ailurus fulgens*)

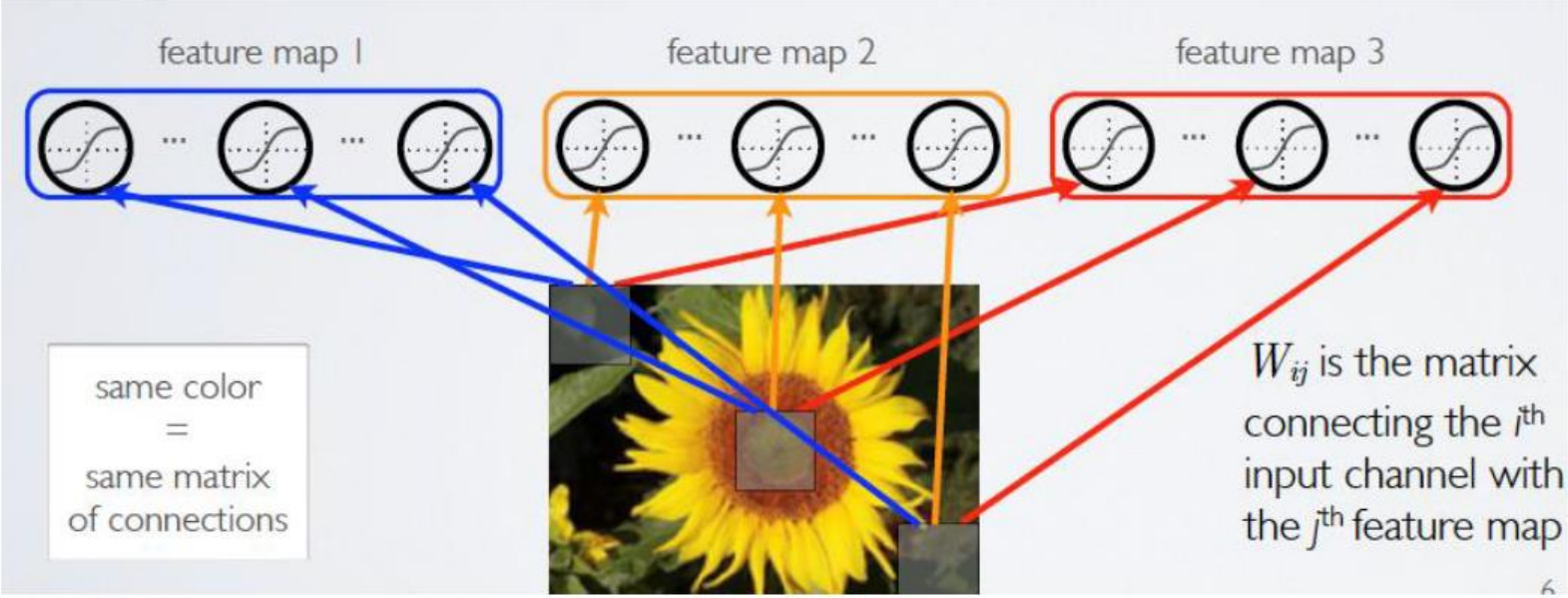
Overview of CNNs

- First idea: Use a local connectivity of hidden units
  - Each hidden unit is connected only to a subregion (patch) of the input image
  - Usually it is connected to all channels
  - Each neuron has a local receptive field



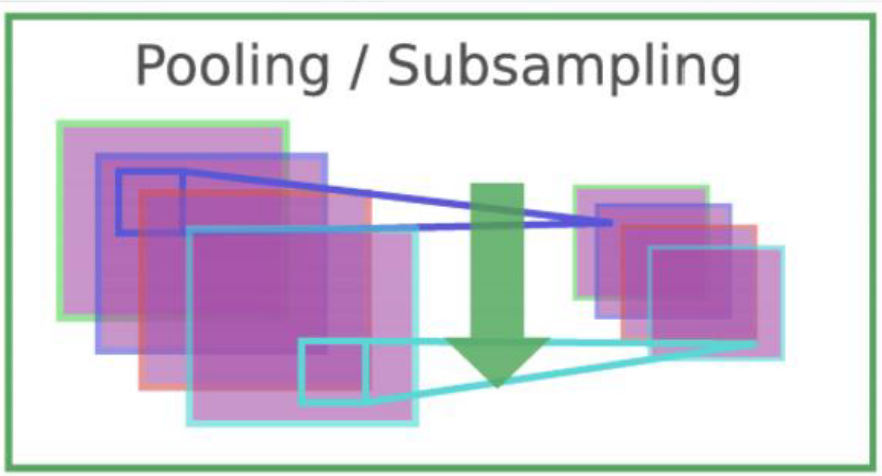
Overview of CNNs

- Second idea: share weights across certain units
  - Units organized into the same "feature map" share weight parameters
  - Hidden units within a feature map cover different positions in the image



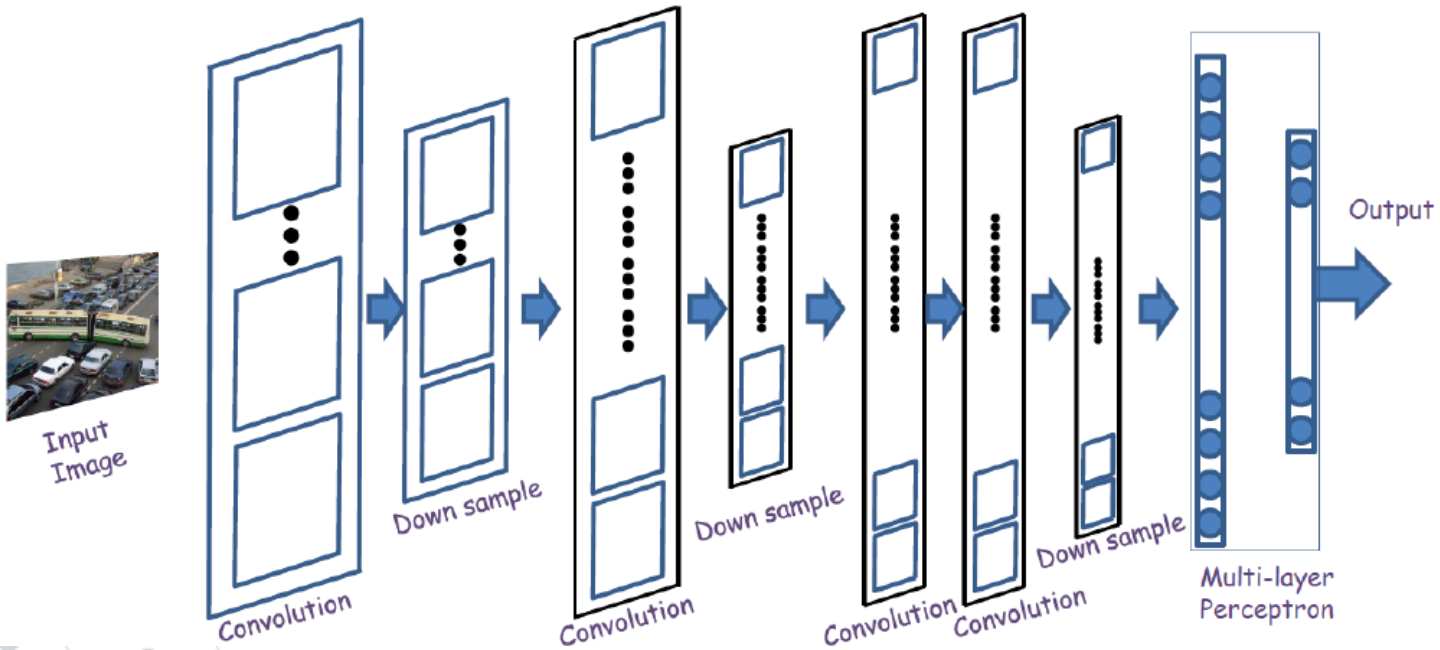
Overview of CNNs

- Third idea: pool hidden units in the same neighborhood
  - Averaging or Discarding location information in a small region
  - Robust toward small deformations in object shapes by ignoring details.



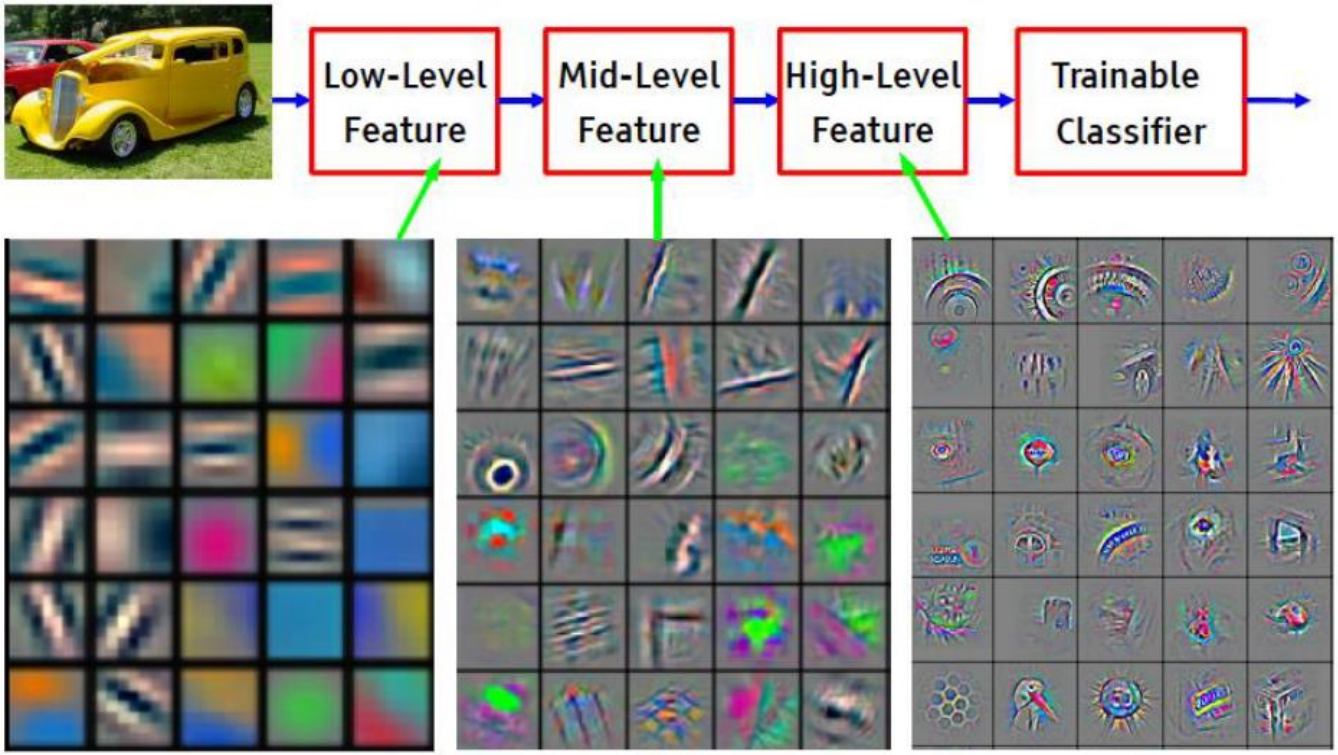
Overview of CNNs

- Fourth idea: Interleaving feature extraction and pooling operations
  - Extracting abstract, compositional features for representing semantic object classes



Overview of CNNs

- Artificial visual pathway: from images to semantic concepts (Representation learning)



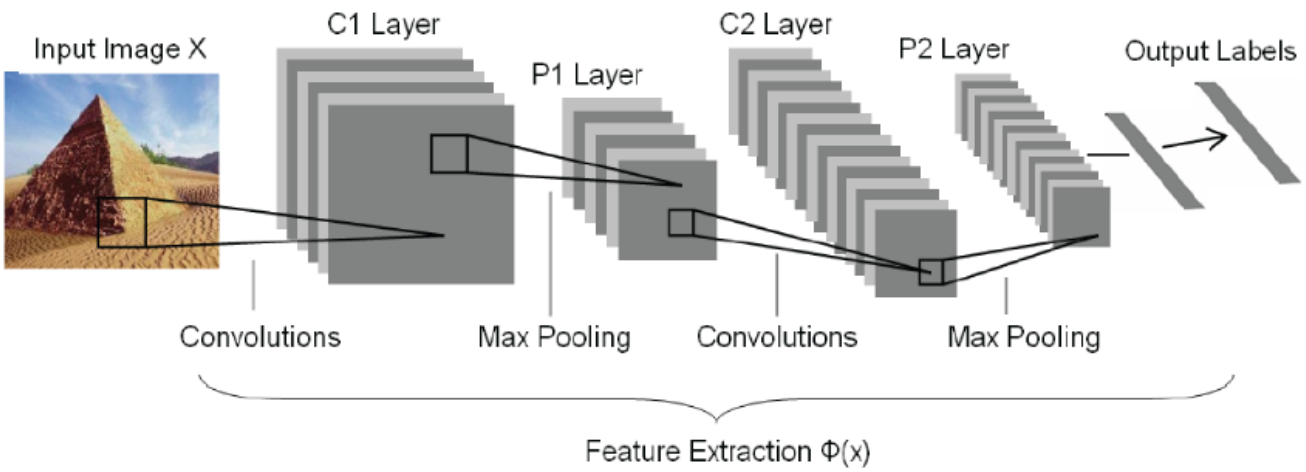
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Convolutional Neural Networks-CNN

- Overview
- Architecture
- Optimization and Regularization
- Practice in Python
- Applications

Convolutional Neural Nets for Image Recognition

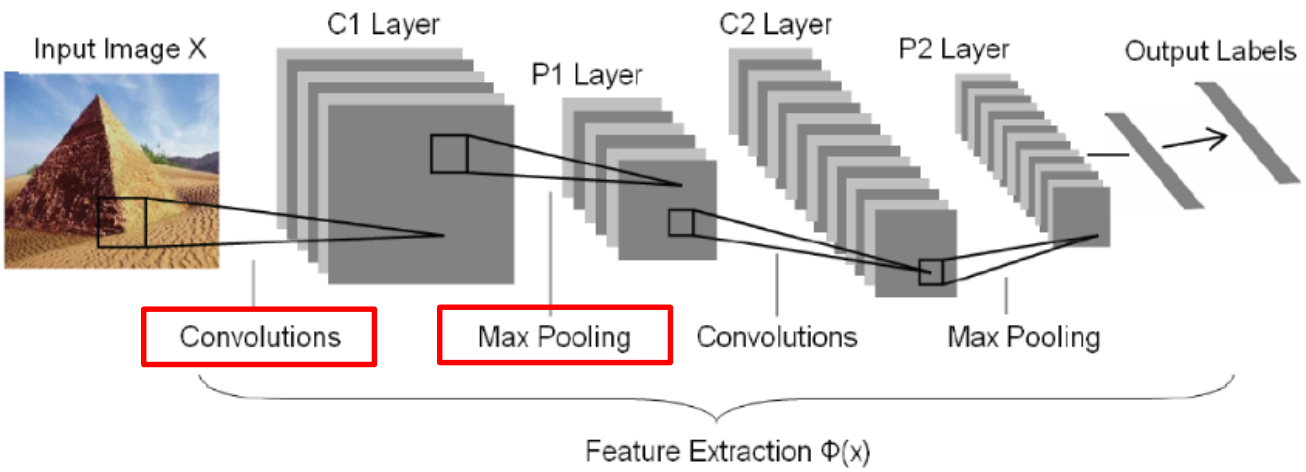
[Le Cun, 1992]



- specialized architecture: mix different types of units, not completely connected, motivated by primate visual cortex
- many shared parameters, stochastic gradient training
- very successful! now many specialized architectures for vision, speech, translation, …

Convolutional Neural Nets for Image Recognition

[Le Cun, 1992]



- specialized architecture: mix different types of units, not completely connected, motivated by primate visual cortex
- many shared parameters, stochastic gradient training
- very successful! now many specialized architectures for vision, speech, translation, …

The Problem With Feed Forward Neural Networks

Take a 256x256 image, that is 65536 parameters

Add a hidden layer with 1000 neurons -> over 65 million parameters

Add two hidden layers with 15000 neurons -> over 1 billion parameters

Call this the curse of dimensionality!

We need a way to represent this data with lower dimensionality, but without losing important information.

The Problem With Feed Forward Neural Networks

In the sensor problem, we took the signal within a **time frame** and used convolutions to output a more desirable representation of it.

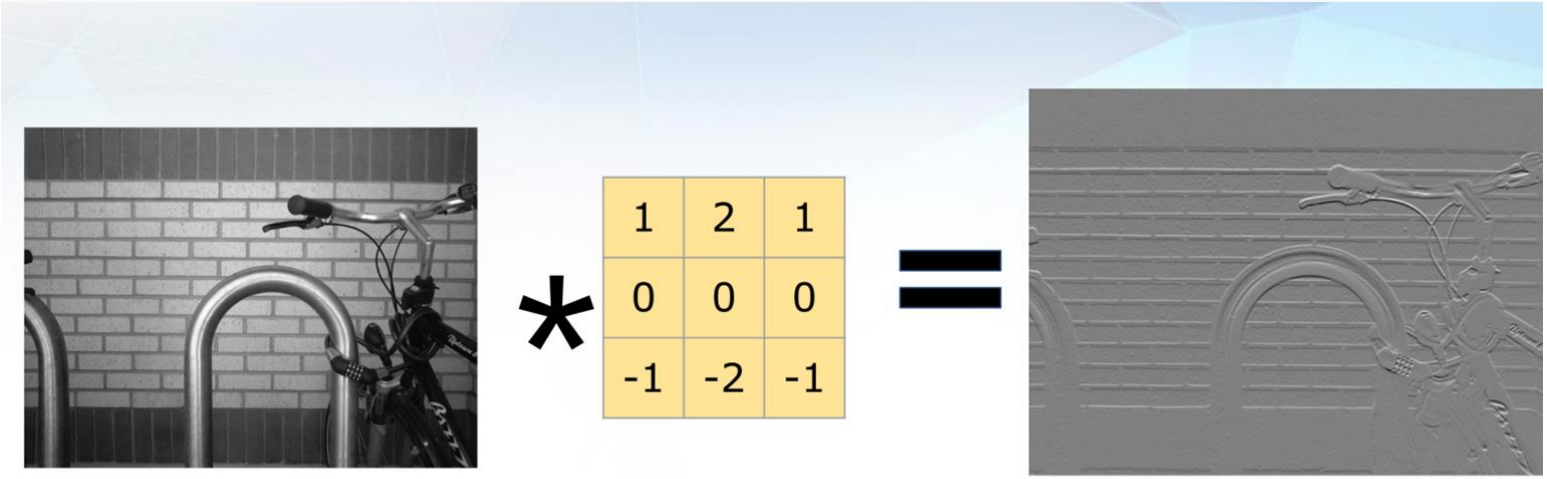
Similarly in images, we can take a **neighbourhood of pixels** and use convolutions to output a more desirable representation of it too.

In the image context, a desirable representation is one that has minimal information loss and one which can also be easily represented in a lower dimension.

Using **convolutions** in such a way is what forms the the basis of **convolutional neural networks (CNNs).**

Convolutional Neural Networks (CNNs)

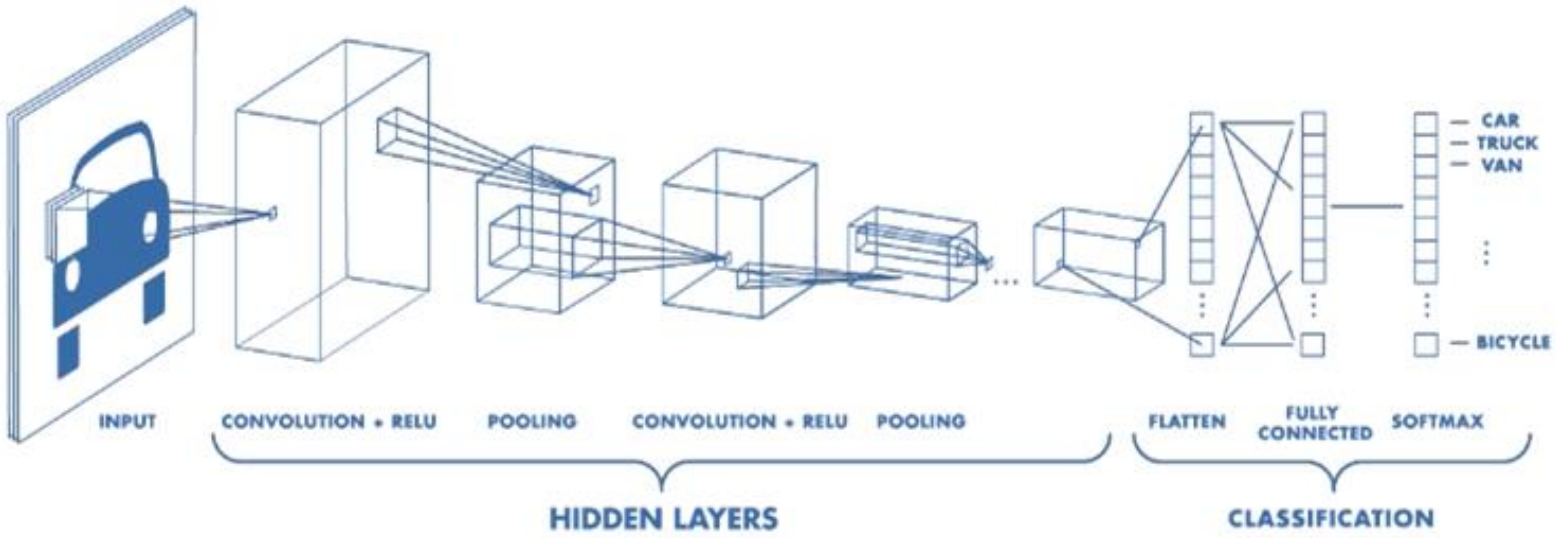
In CNNs, we apply convolutional filters to an image to consider the **neighborhood of inputs before considering the correlation of that pixel with those on the other side of the image.**



CNNs - The Big Picture

Convolutional neural networks have 3 stages:

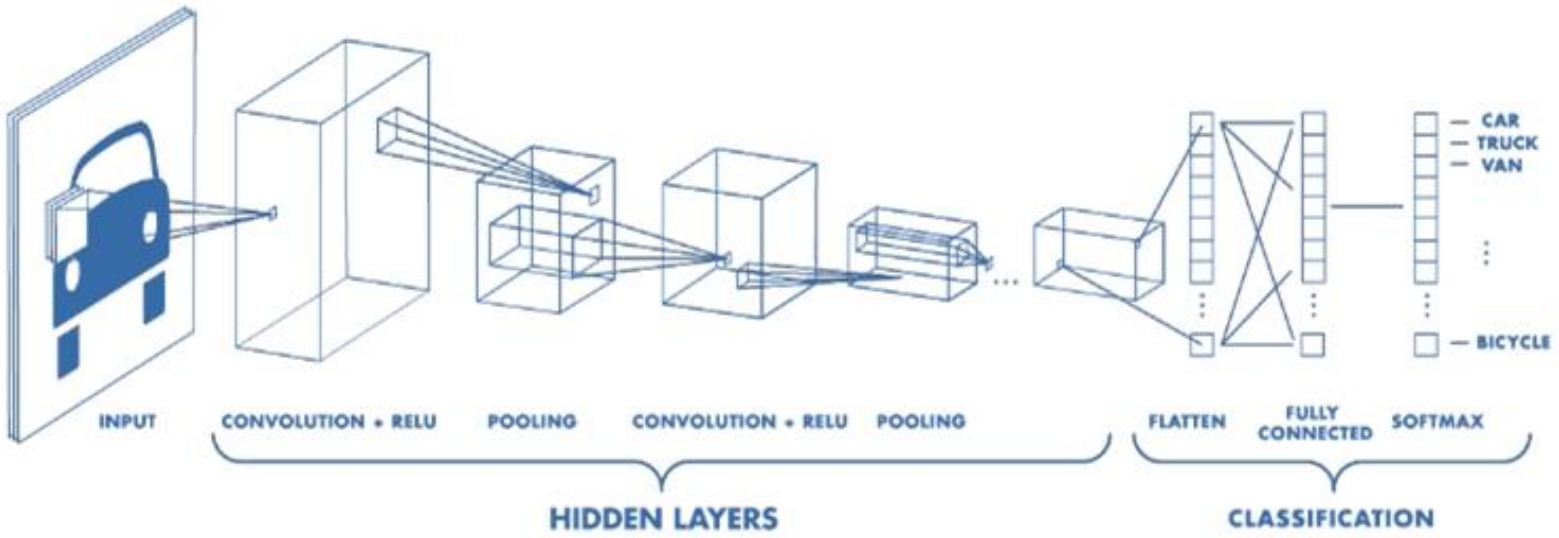
1. Convolution Layer(s)
2. Pooling Layer(s)
3. Classification - Fully Connected Layer



CNNs - The Big Picture

Convolutional neural networks have 3 stages:

1. **Convolution Layer(s)**
2. Pooling Layer(s)
3. Classification - Fully Connected Layer

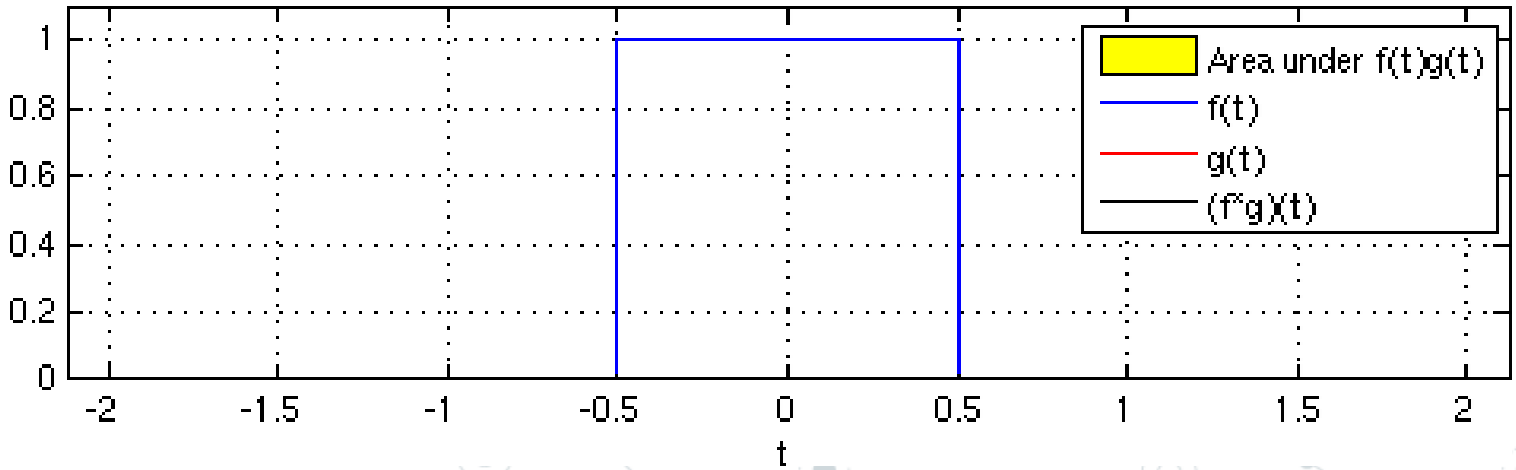


Convolutions

A **convolution** is a mathematical operation on two functions which produces a third one.

It can be calculated with the following mathematical equation:

$$s(t) = \int f(a)g(t-a)da \quad \longleftrightarrow \quad s(t)=(f*g)(t)$$

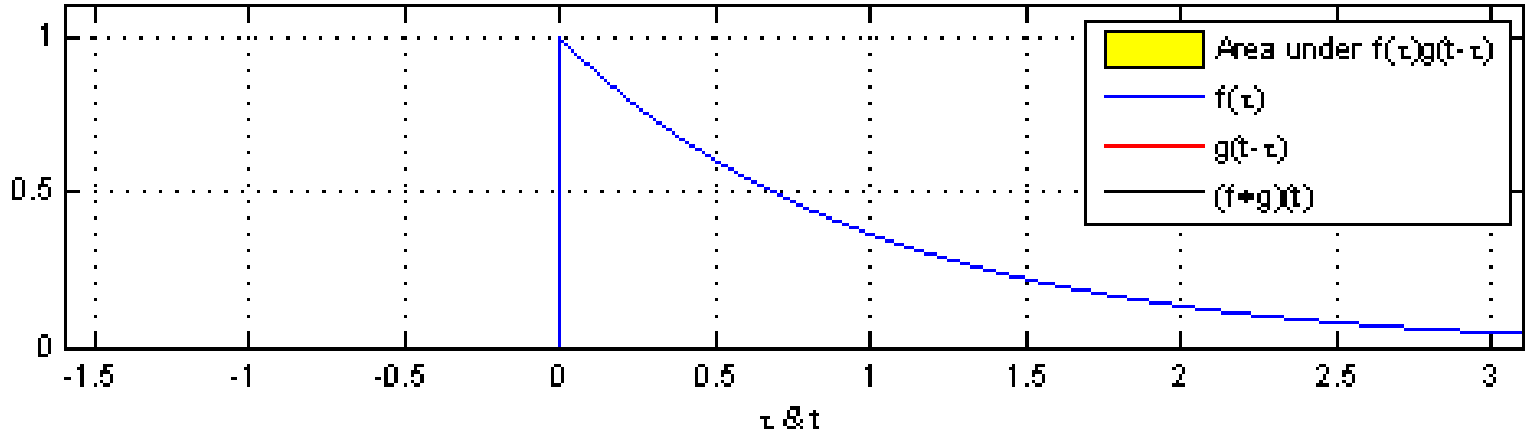


Convolutions

A **convolution** is a mathematical operation on two functions which produces a third one.

It can be calculated with the following mathematical equation:

$$s(t) = \int f(a)g(t-a)da \quad \longleftrightarrow \quad s(t)=(f*g)(t)$$



Convolutions

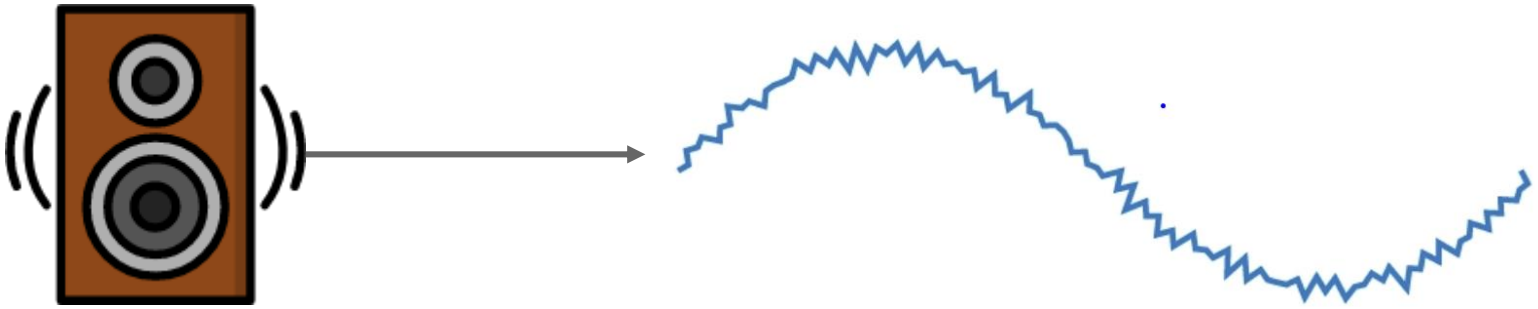
There is also a discrete version of the convolution operation where we replace the integral with the sum.

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{a=\infty} x(a)w(t-a)$$

Sensor Problem

Say we have a sensor that outputs a noisy signal.

This signal is time dependent -> ***x(t)***

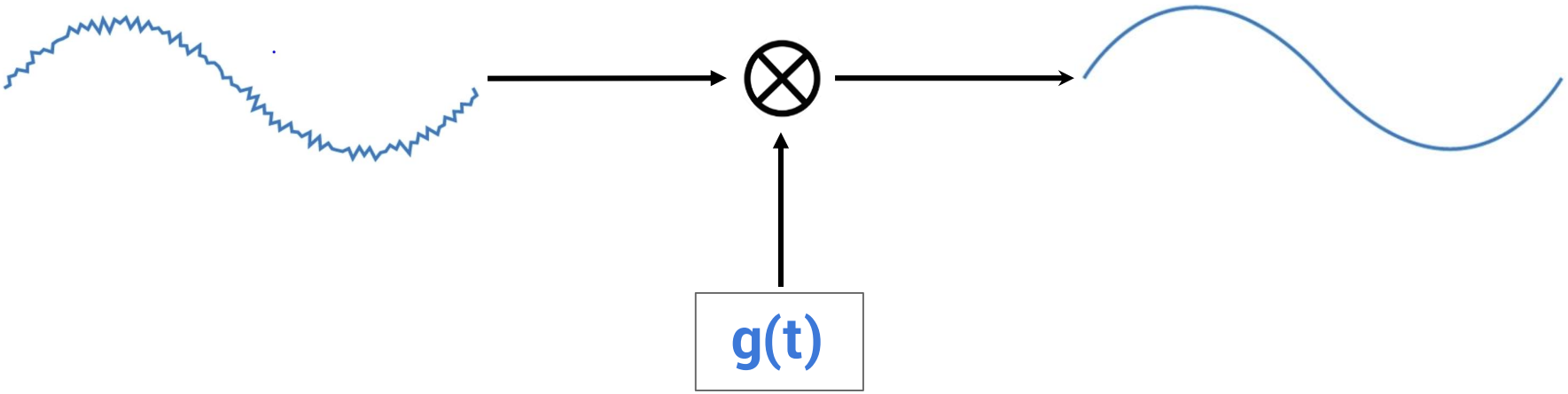


Convolutions can be used to smooth out this signal. How?

Sensor Problem

We can **convolute** a signal with our original one to obtain a new smoothened signal, with **minimal information loss**.

Note that this method takes a **weighted average** of the most recent data at each point in order to denoise the signal.

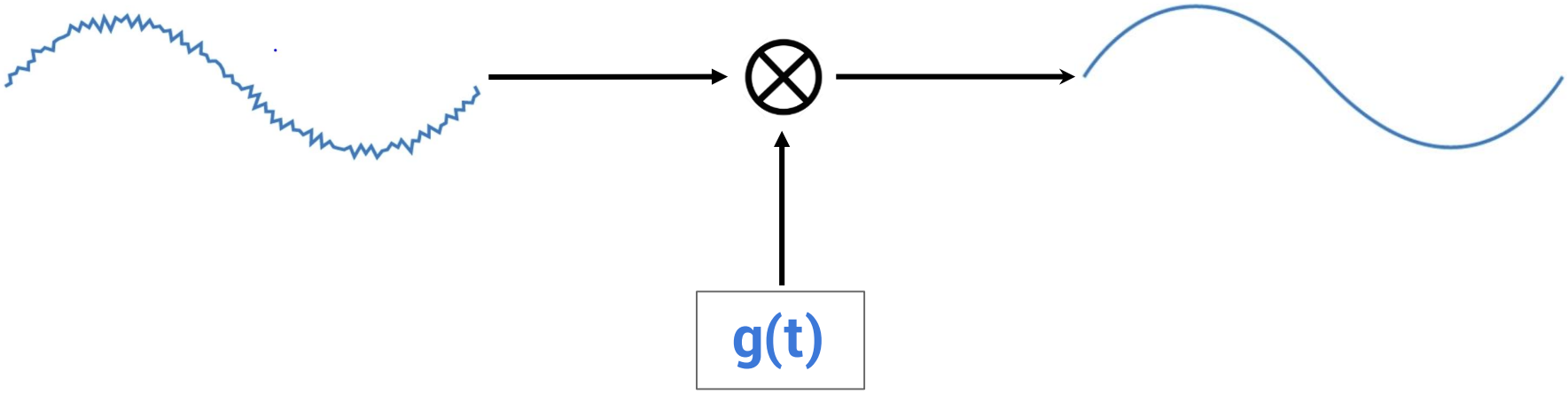


Sensor Problem

Before, we needed a very complex function to represent this signal.

Now, we can use a simple function (such as sine or cosine).

But how is this related to machine learning ?!?!



CNNs - Convolution Layer(s)

The goal of a convolution layer is to **extract the high level features** out from an image using convolutional filters, called the **kernel**.

A 3x3 kernel is usually used, but a 5x5 and 7x7 can also work depending on applications. This is a **hyperparameter** to explore.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

1	0	1
0	1	0
1	0	1

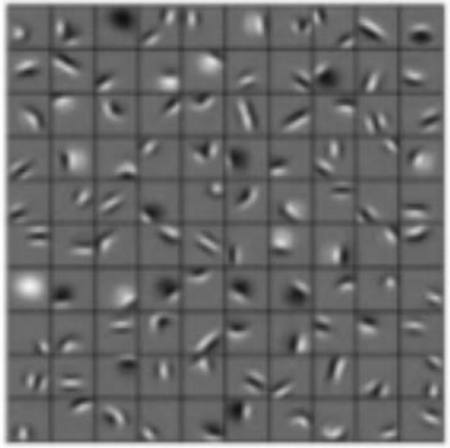
Filter / Kernel

CNNs - Convolution Layer(s)

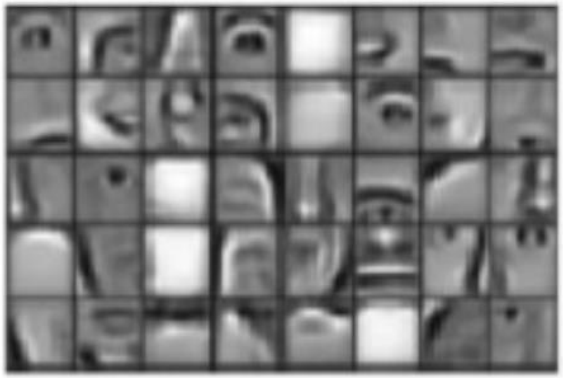
There can be multiple convolution layers where earlier layers captures low-level features such as edges or colours while added layers capture high-level features such as facial parts.

Below are examples of **feature maps** created from a convolution layer.

Low-level feature



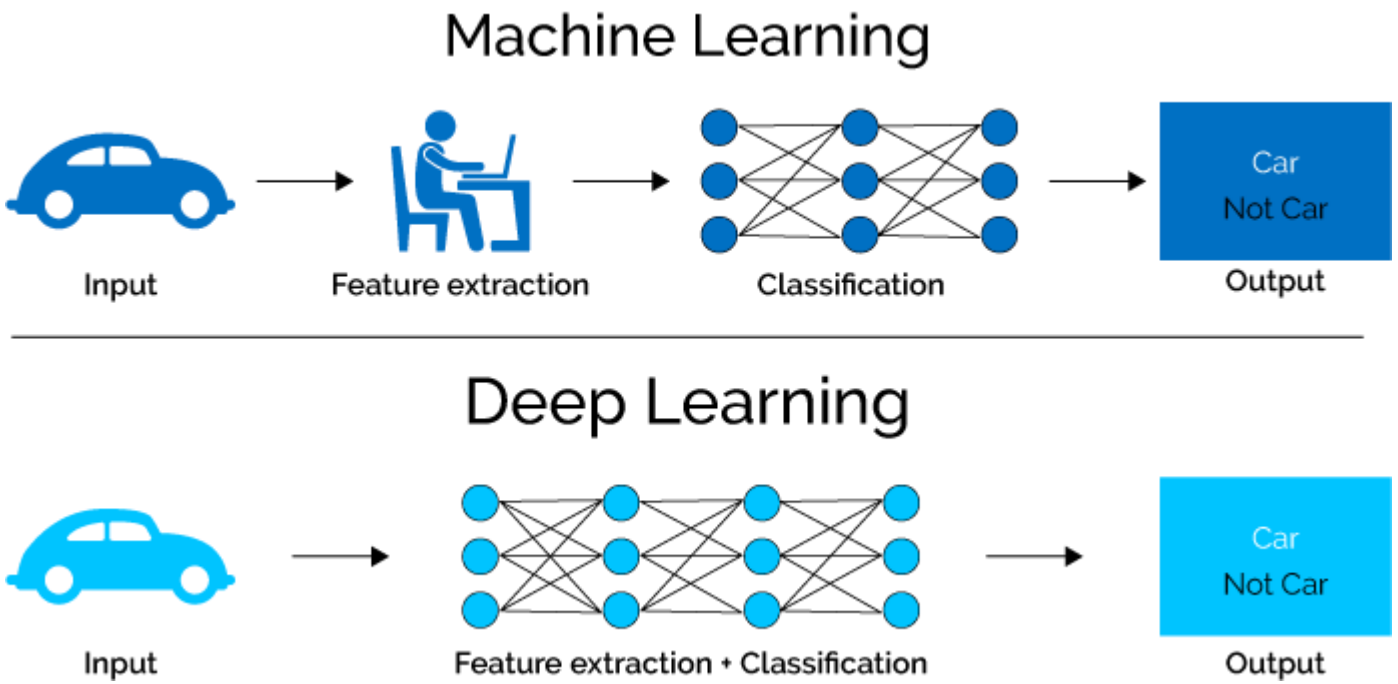
Mid-level feature



High-level feature

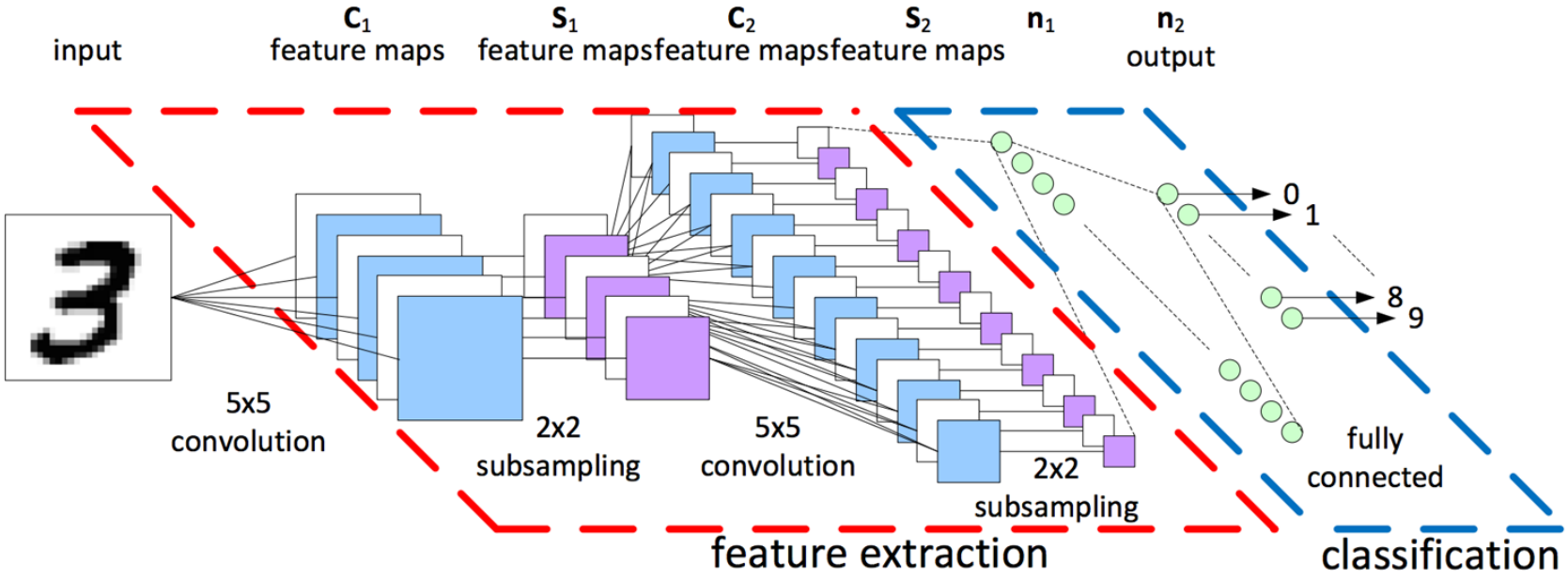
CNNs - Convolution Layer(s)

In CNNs and deep learning in general, the **features are learned** rather than manually selected during the data preprocessing phase.



CNNs - Convolution Layer(s)

Each convolution layers helps to extract a different set of features.



CNNs - Convolution Layer(s)

Let's visualize exactly how a convolution would work.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

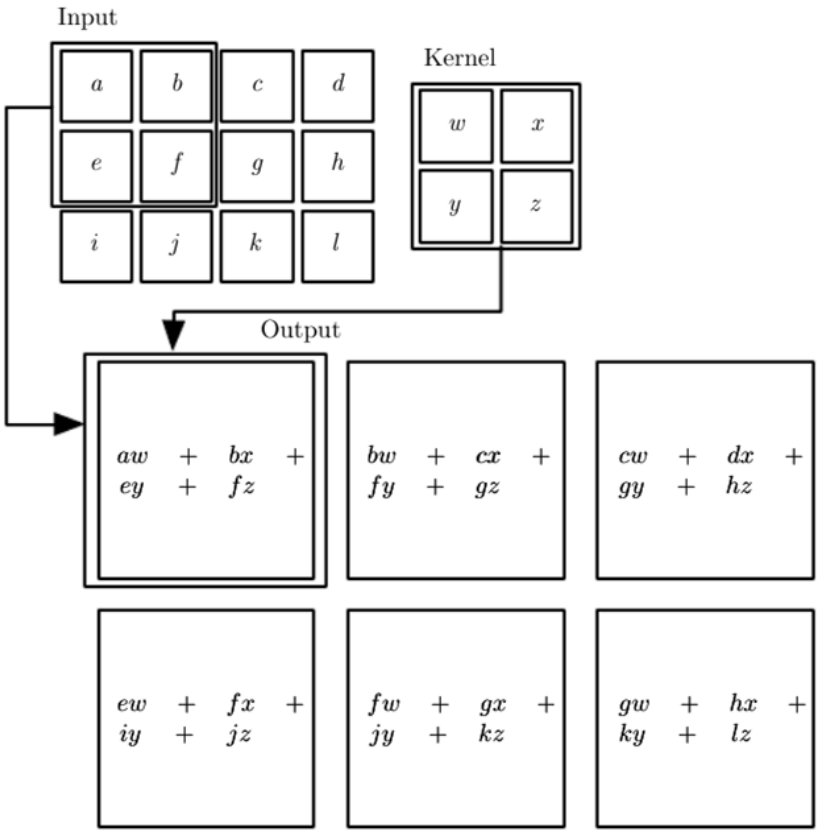
Input

1	0	1
0	1	0
1	0	1

Filter / Kernel

CNNs - Convolution Layer(s)

Let's visualize exactly how a convolution would work.



CNNs - Convolution Layer(s)

Let's visualize exactly how a convolution would work.

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

Input x Filter

4		

Feature Map

CNNs - Convolution Layer(s)

Let's visualize exactly how a convolution would work.

1	1x1	1x0	0x1	0
0	1x0	1x1	1x0	0
0	0x1	1x0	1x1	1
0	0	1	1	0
0	1	1	0	0

Input x Filter

4	3	

Feature Map

CNNs - Convolution Layer(s)

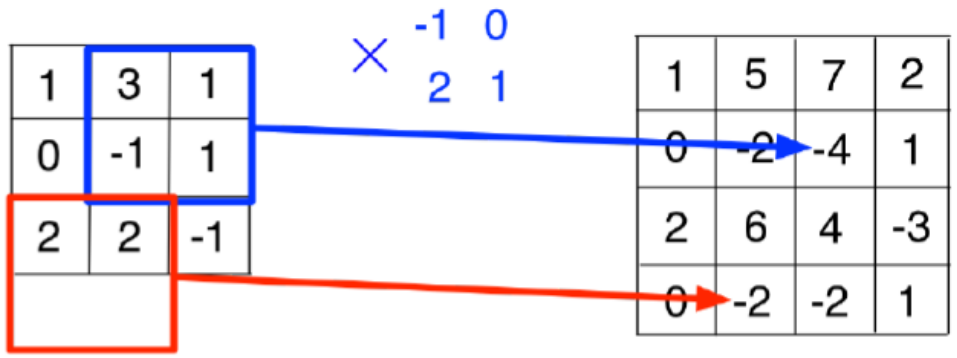
If A and B are two 2-D arrays, then:

$$(A * B)_{ij} = \sum_s \sum_t A_{st} B_{i-s,j-t}.$$

1	3	1
0	-1	1
2	2	-1

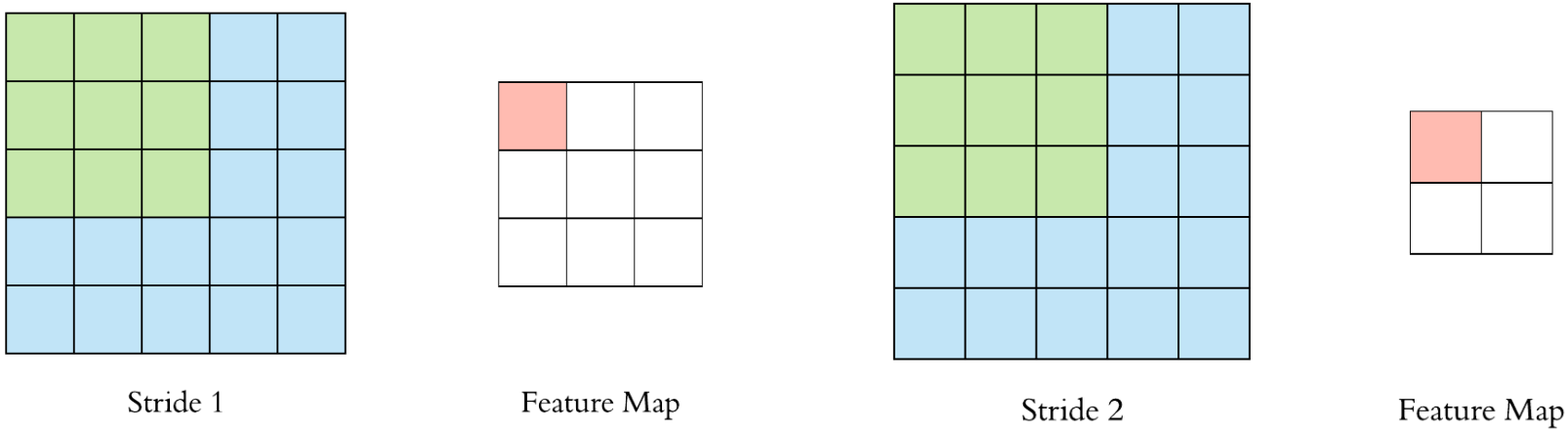
$*$

1	2
0	-1



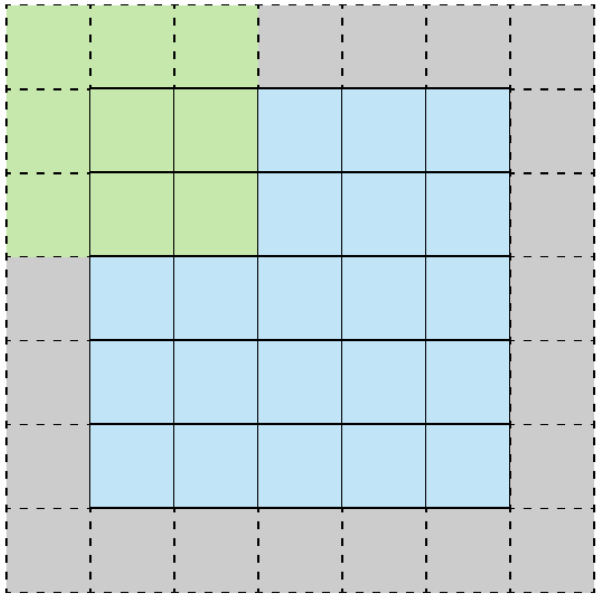
CNNs - Convolution Layer(s)

We can change the **stride** of the convolution depending on whether we want more or less overlap. (Usually use stride = 1)

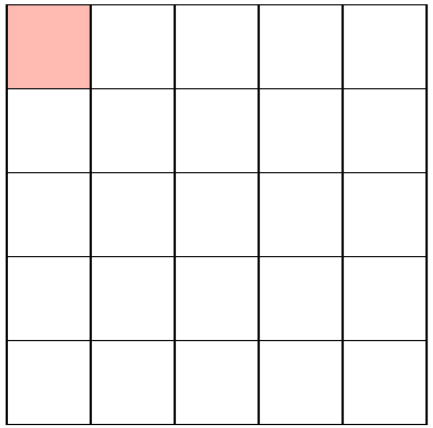


CNNs - Convolution Layer(s)

We usually **padding** (0 pixels) to the outside of the image to keep the input size the same as its feature map.



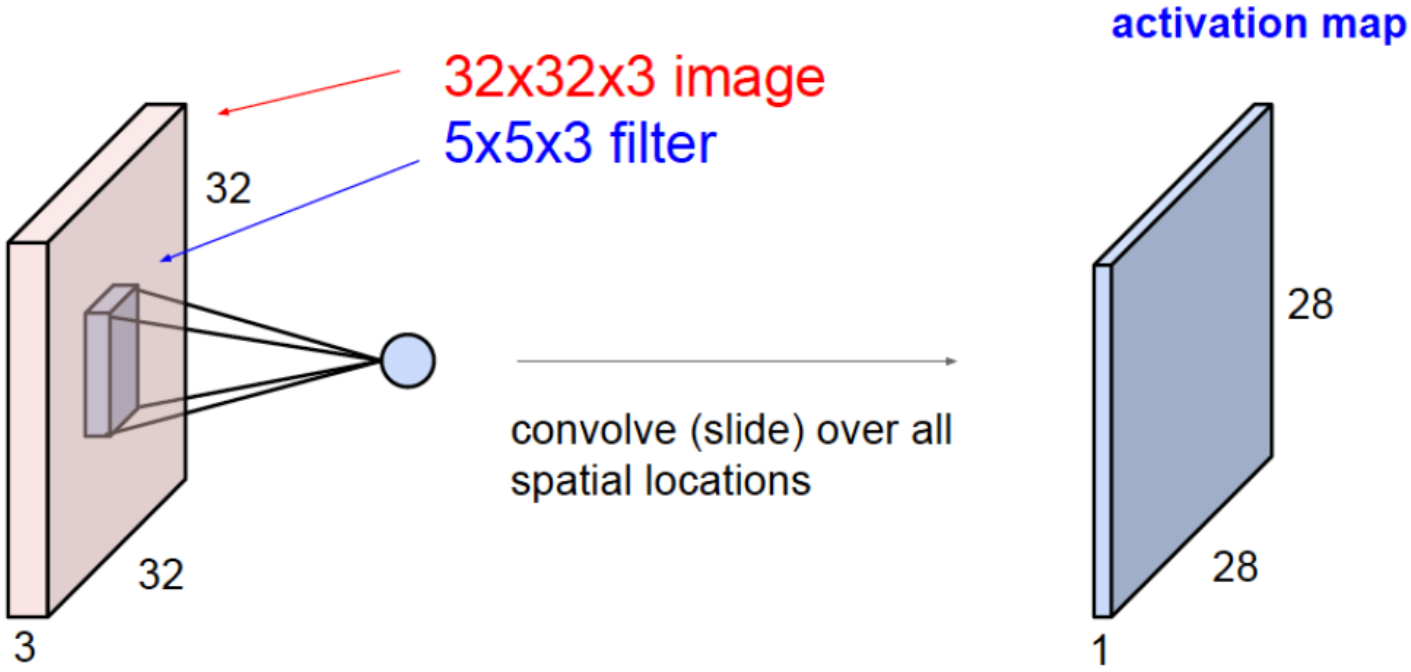
Stride 1 with Padding



Feature Map

CNNs - Convolution Layer(s)

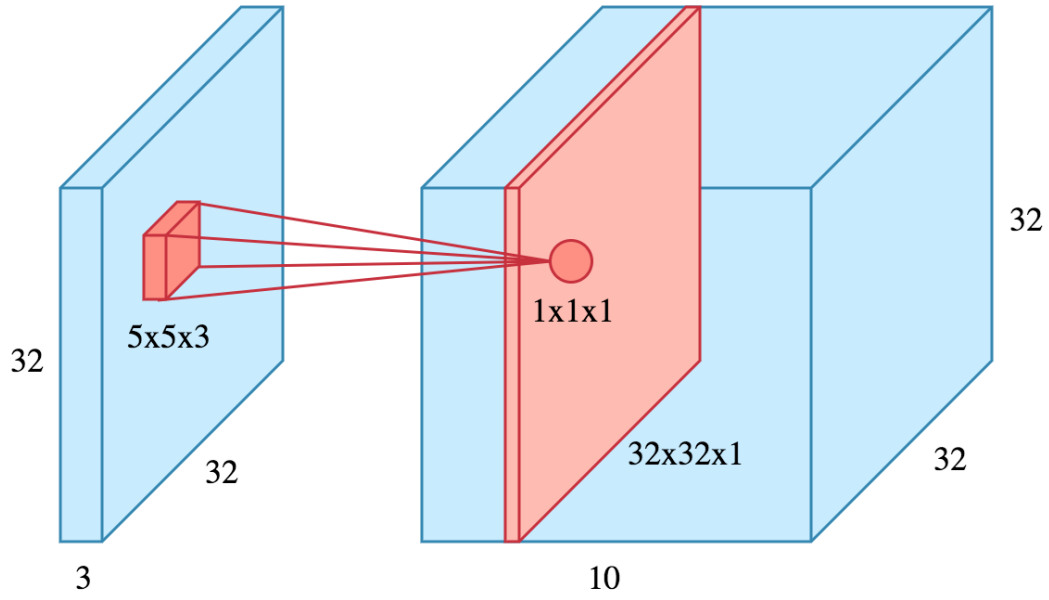
- Convolution operation
  - Parameter sharing
  - Spatial information



CNNs - Convolution Layer(s)

We perform multiple convolutions on an input, each using a different filter **(filter count)** and resulting in a distinct feature map.

We then stack them together as the output of our convolution layer.

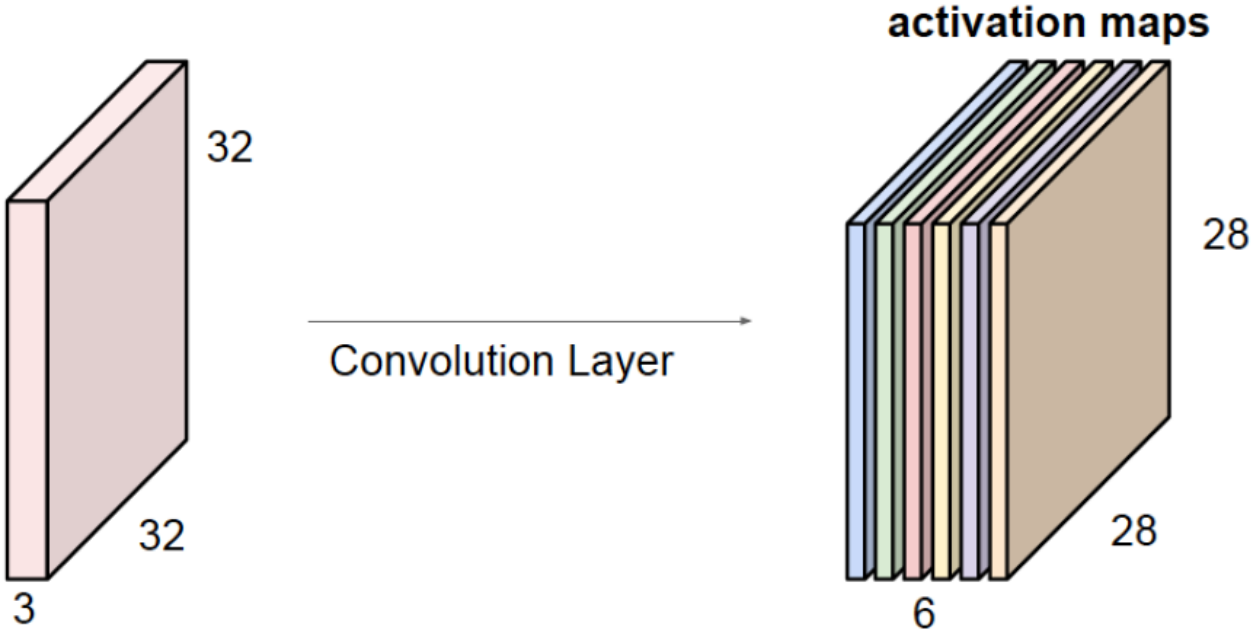


A quick note that the depth of the convolution filter should match the number of channels of the image (RGB = 3 channels).

CNNs - Convolution Layer(s)

- Multiple kernels/filters

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

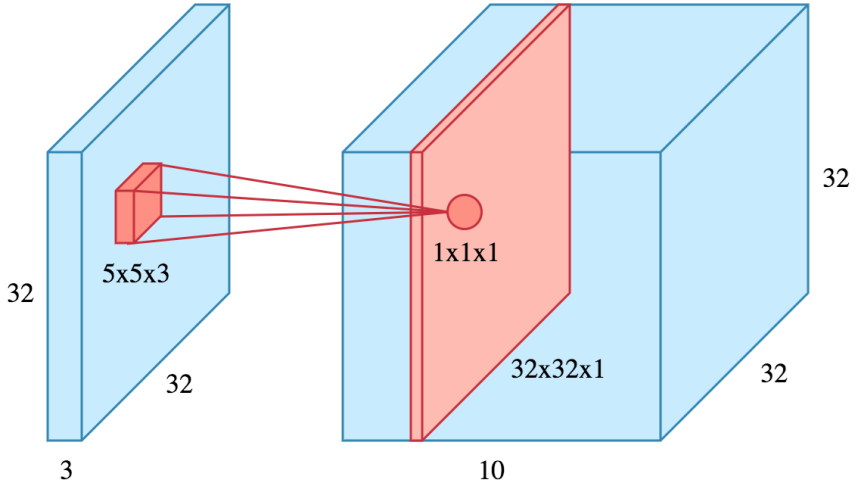


We stack these up to get a "new image" of size 28x28x6!

CNNs - Convolution Layer(s)

Summarizing the **hyperparameters** in a convolution layers:

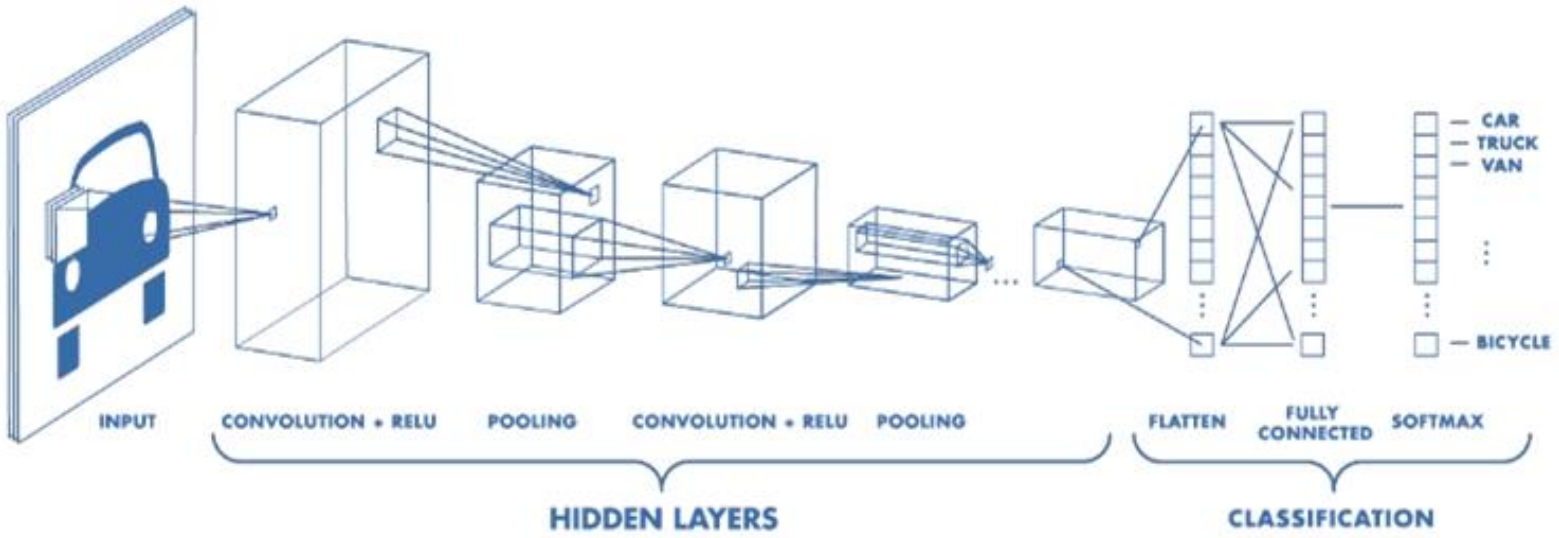
- Filter Size - Usually 3x3, but 5x5 and 7x7 can sometimes work.
- Filter Count - Most variable parameter but is usually a power of 2 between 32 and 1024. High counts are prone to overfitting. The filter count usually increases at deeper layers.
- Stride - Usually 1.
- Padding - Usually enabled.



CNNs - The Big Picture

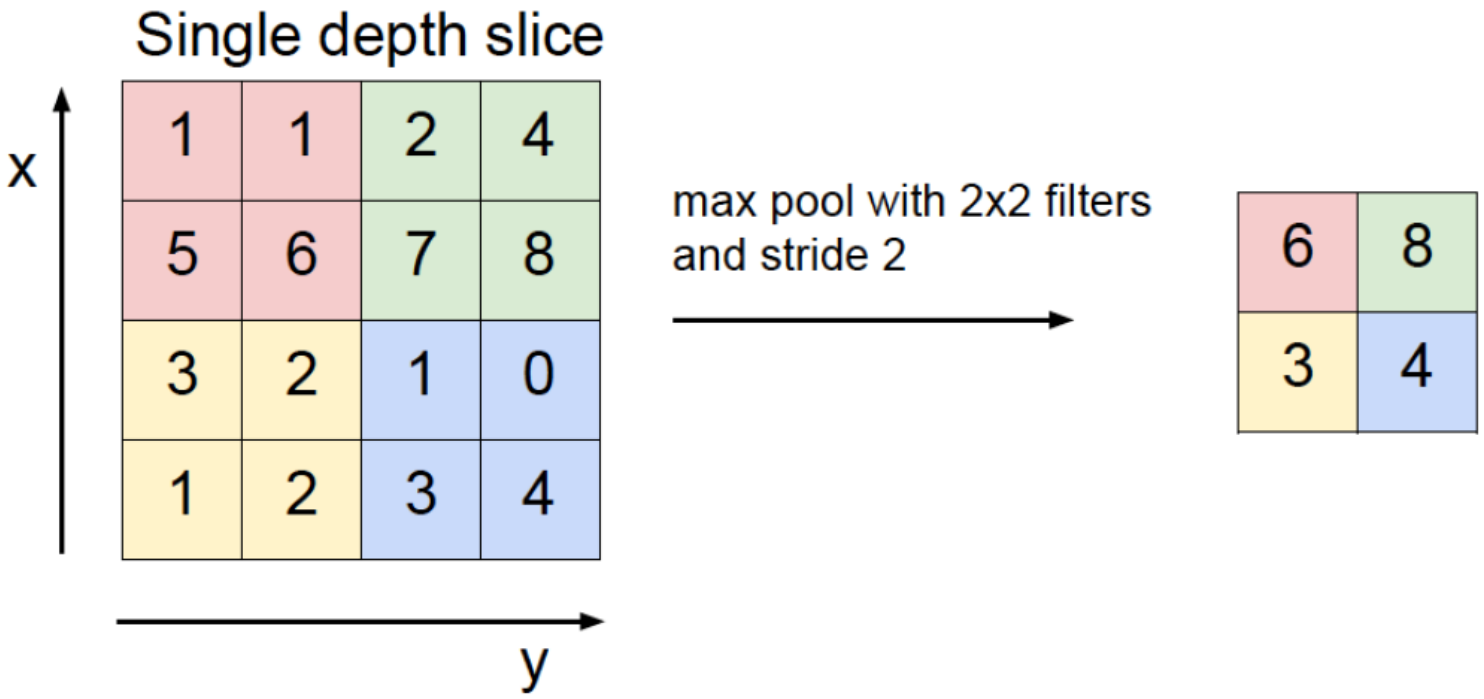
Convolutional neural networks have 3 stages:

1. Convolution Layer(s)
2. **Pooling Layer(s)**
3. Classification - Fully Connected Layer



CNNs - Pooling Layer(s)

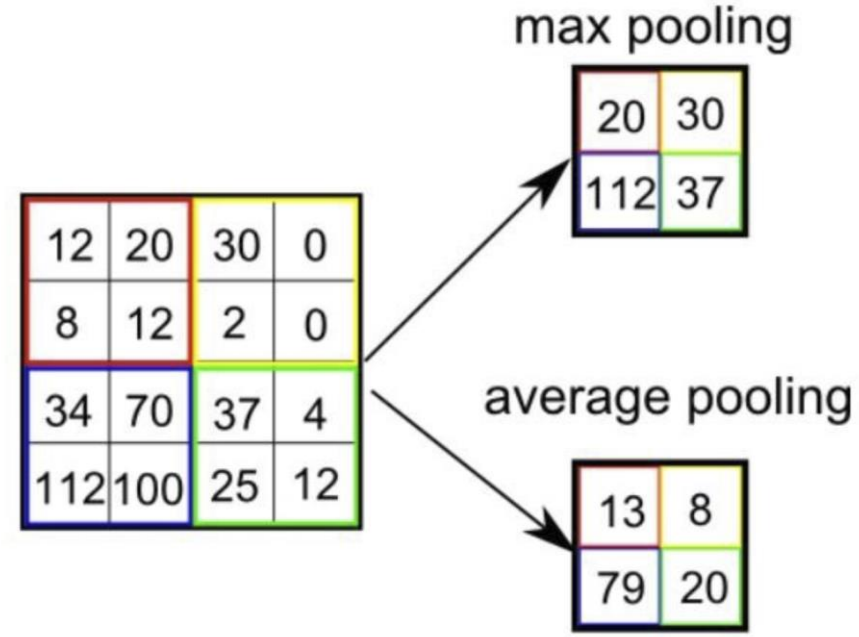
- Example: max pooling



CNNs - Pooling Layer(s)

The goal of a pooling layer is to reduce the spatial size of the convolved feature. There are two main types of pooling layers:

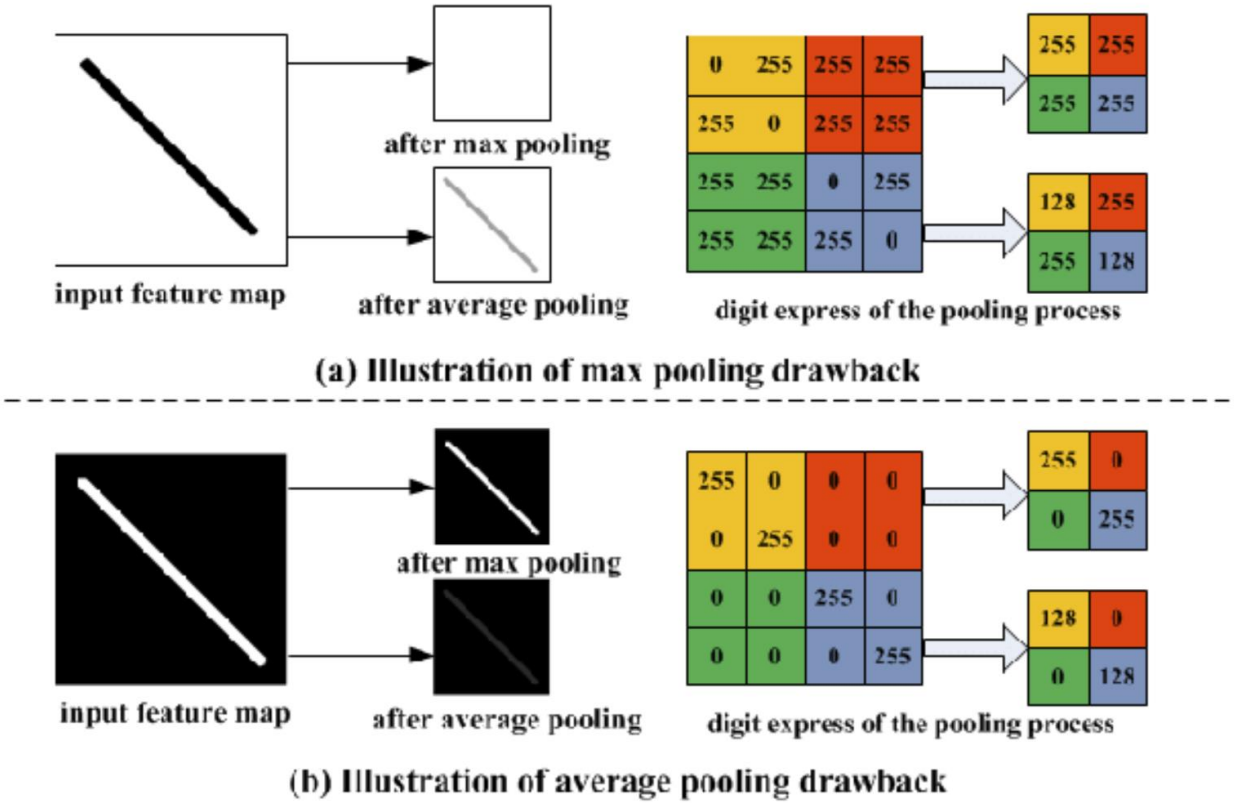
- **Max pooling** returns the maximum value from the portion of the image covered by the kernel.
- **Average pooling** returns the average of all values from the portion of the image covered by the kernel.
- **L2 norm** and **weighted average** are other types.



Note that the size of the pooling window is another hyperparameter.

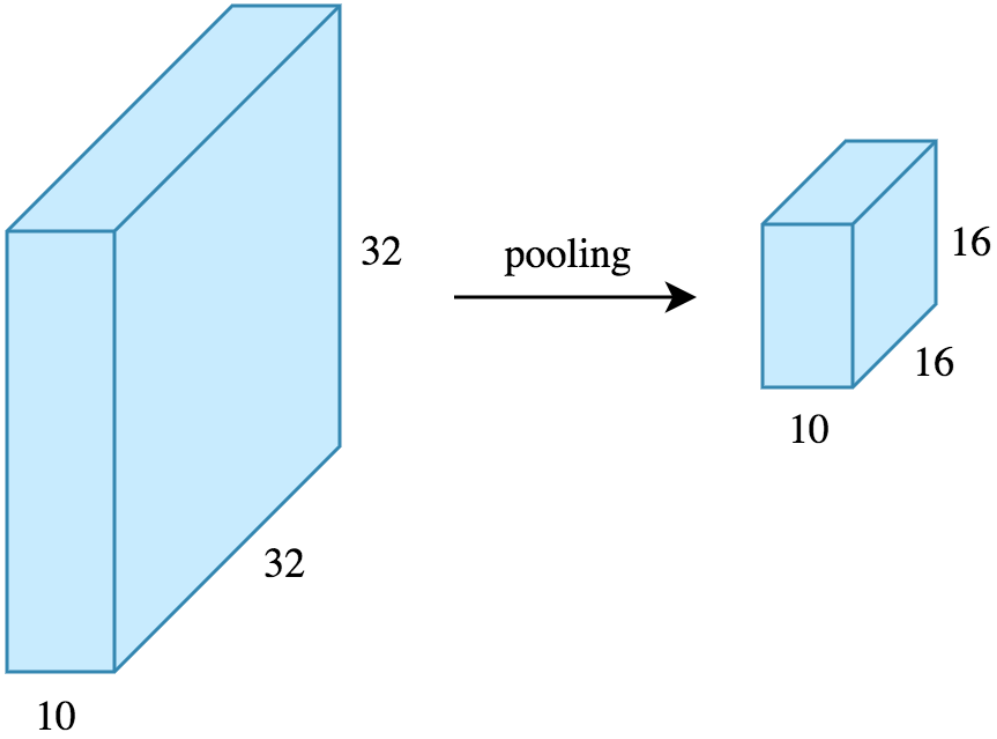
CNNs - Pooling Layer(s)

Max pooling has shown better empirical results, but the type of pooling layer that you use depends on the scenario:



CNNs - Pooling Layer(s)

While pooling reduces the height and width of the feature map, it does not change the depth as pooling performs independently on each depth.



CNNs - Pooling Layer(s)

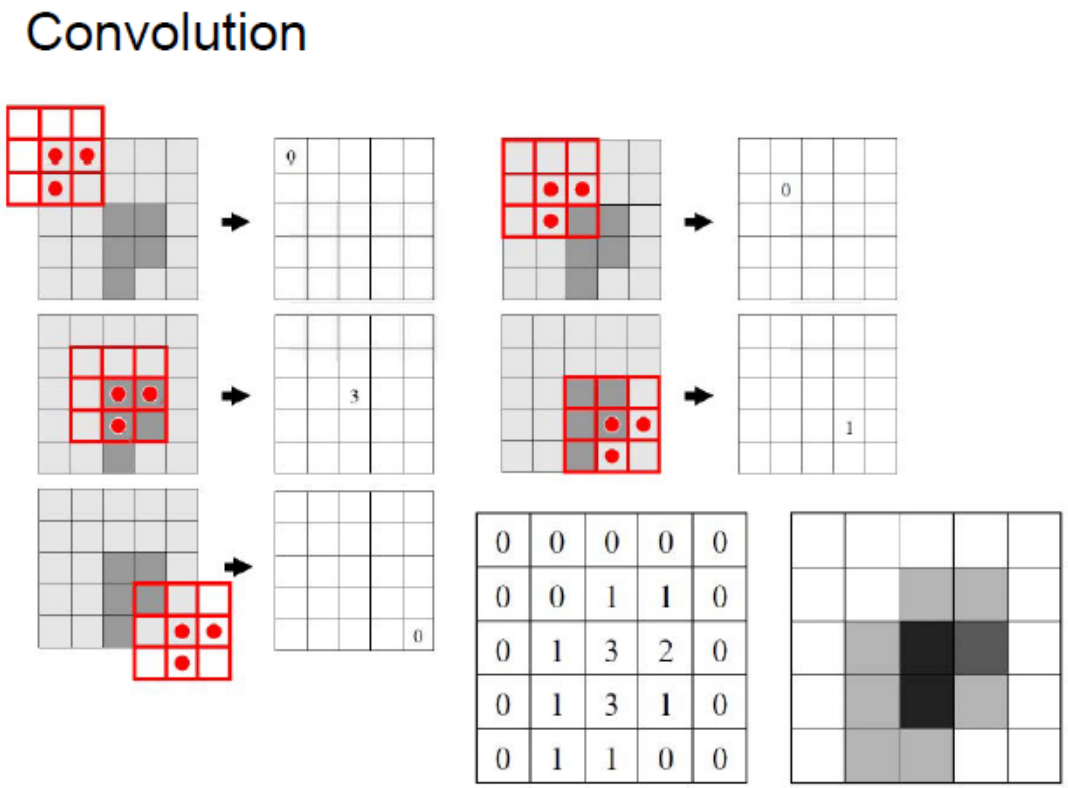
This helps to reduce the number of dimensions which **shortens training time** and **prevents overfitting**.

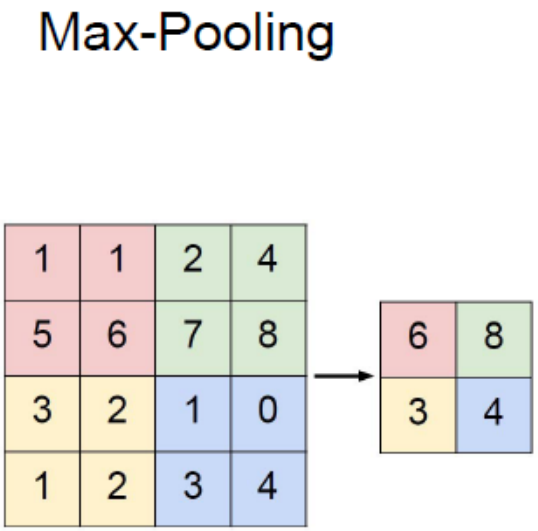
Dominant features are extracted and thus maintaining the process of effectively training the model.

Essentially the pooling layer takes an area and performs a statistical summary of it.

This allows us to achieve the goal of reducing the number of dimension, but without losing important information.

CNNs – Example of Convolution and Pooling

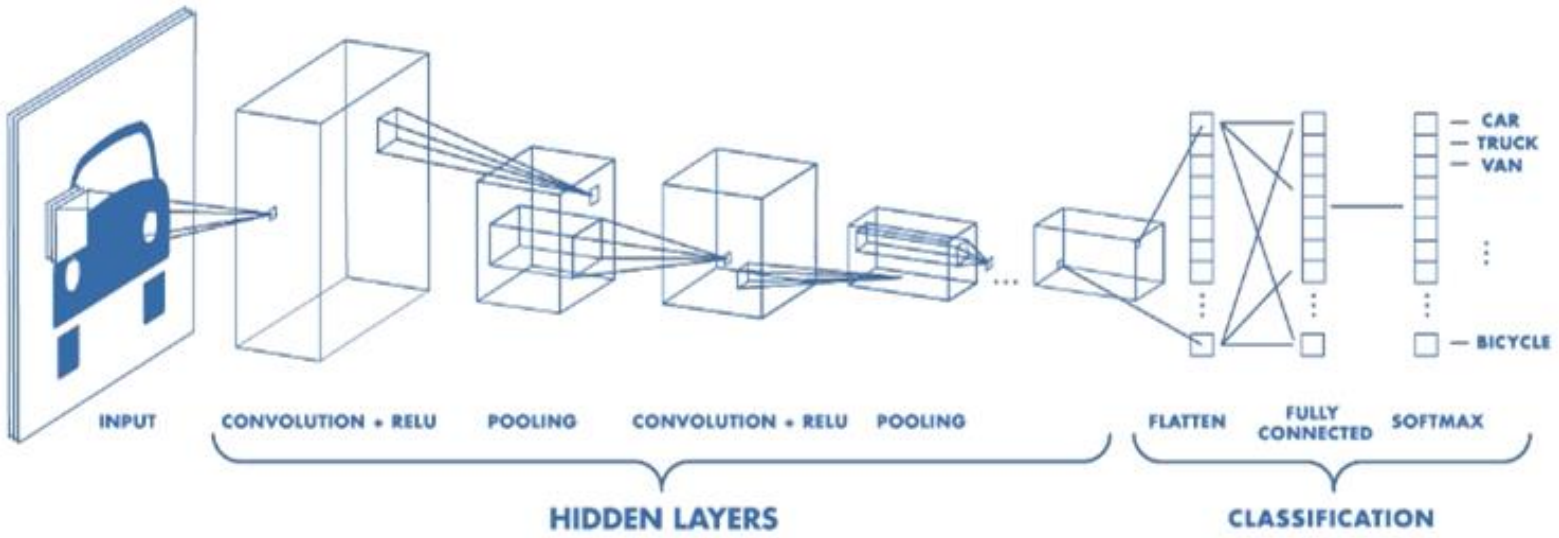




CNNs - The Big Picture

Convolutional neural networks have 3 stages:

1. Convolution Layer(s)
2. Pooling Layer(s)
3. **Classification - Fully Connected Layer**

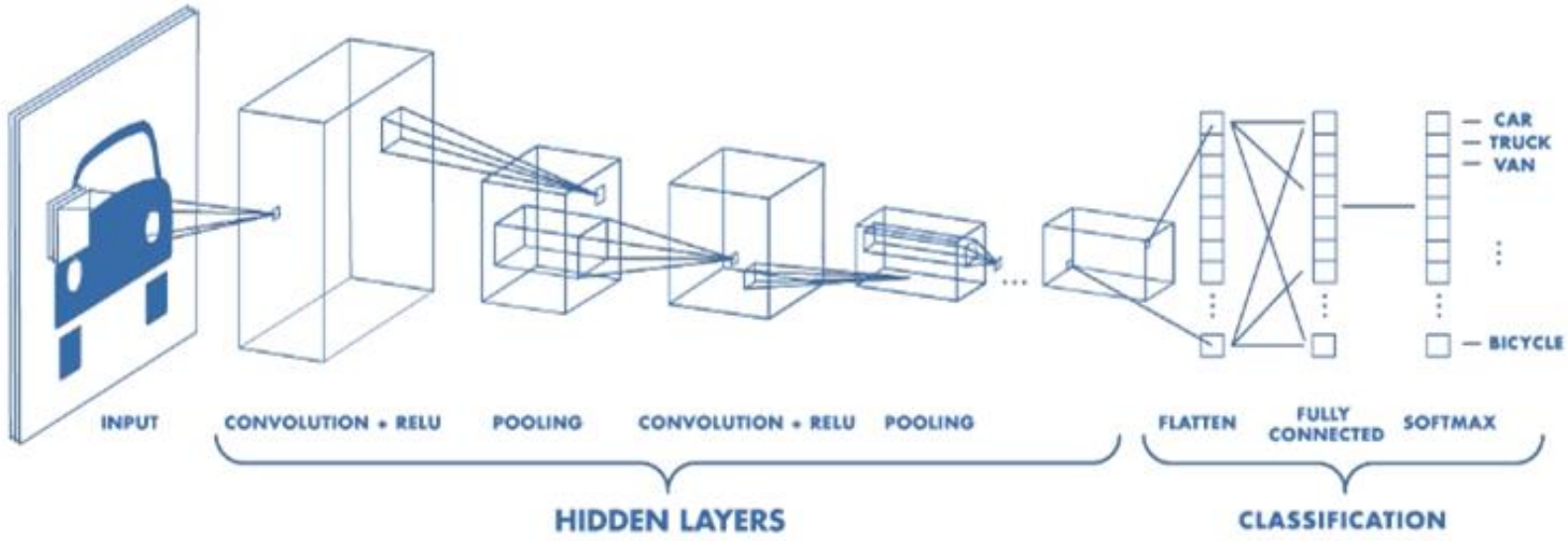


CNNs - Fully Connected Layer

The reduced form of our image is flattened into a column vector and is fed through a feed forward neural network.

The training happens the same way (using backpropagation and gradient descent), but the math is just more complex.

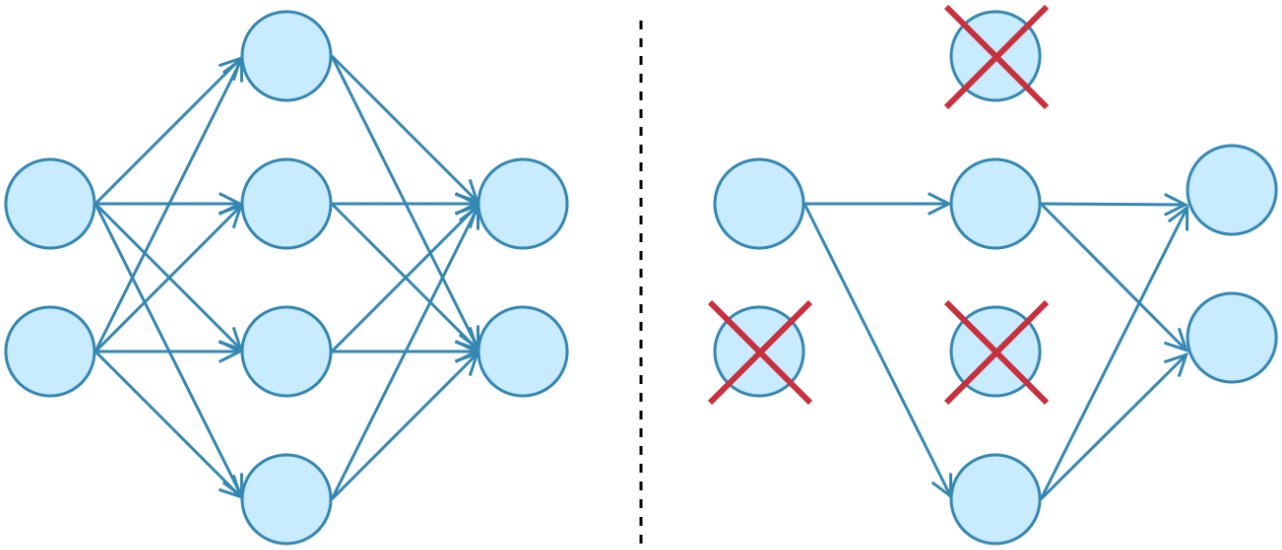
In depth explanation if you are interested.



Dropout - Regularization Technique

This is basically the idea of dropout - **disabling neurons with probability p** so that the network isn't dependent on one node.

Dropout can be applied to input or hidden layers, but not output.



No Dropout

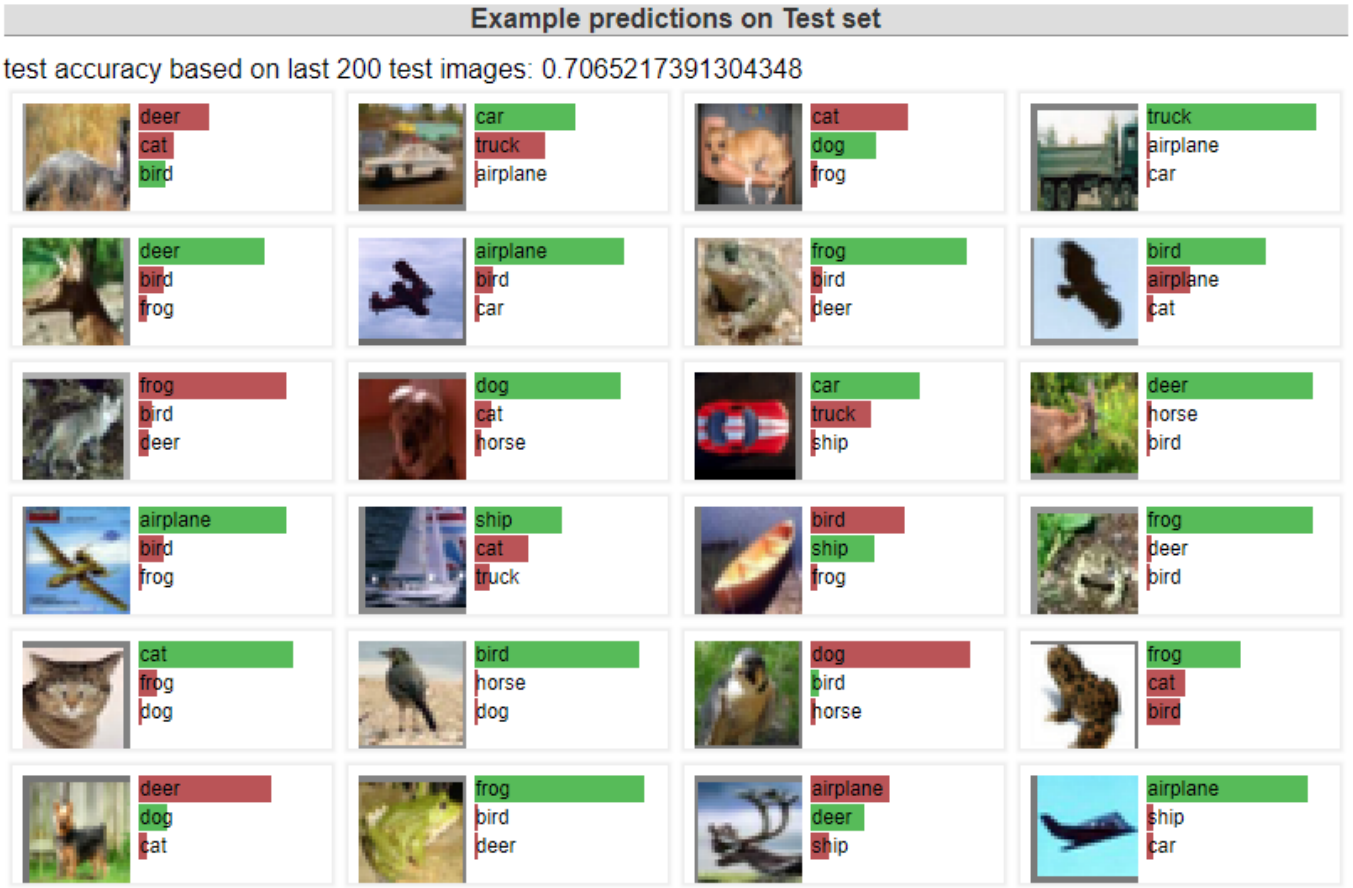
With Dropout

CNN Advantages

Using convolutions give us the following advantages:

- Feature Extraction
  - Automatically captures and learns relevant features.
- Lower Computational Load
  - Reduced dimensionality helps to speed up training time.
- Spatial Invariance
  - Since weights are repeated across space, it does not matter where an object may be located in a photo.
- Performance
  - CNNs are a breakthrough discovery which greatly outperforms traditional ML algorithms in computer vision and other tasks.

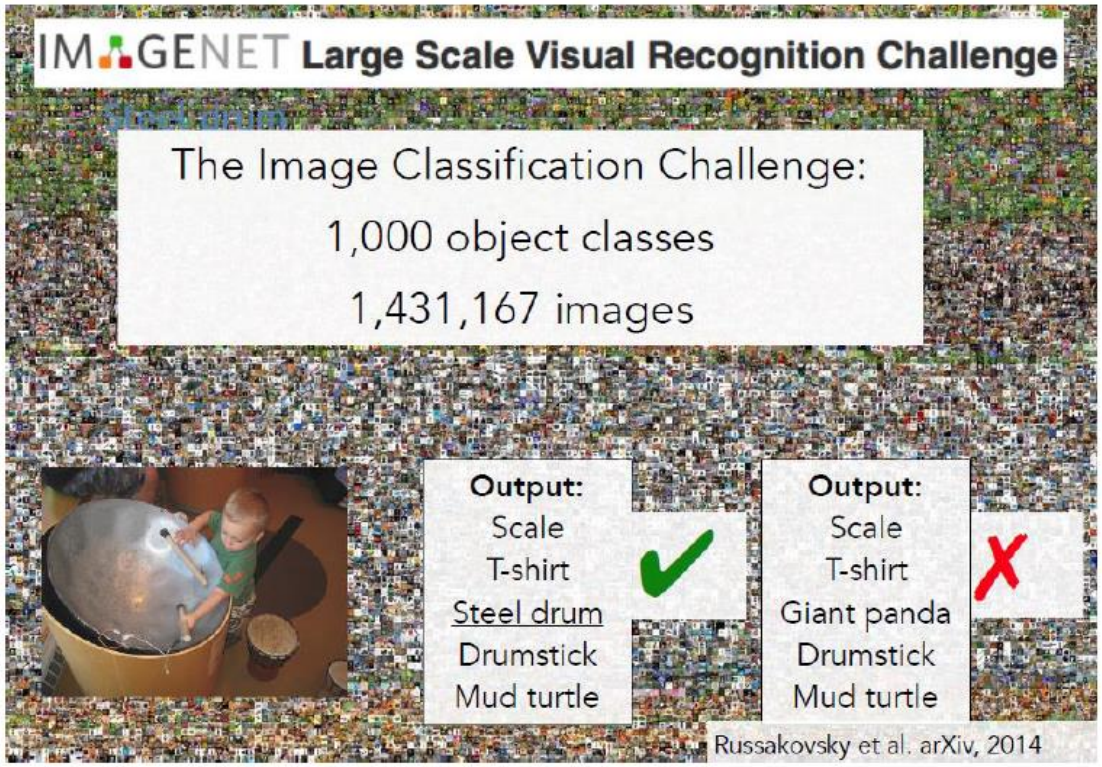
CNNs - Demo



[CNN - demo]

Background: Image/Object Classification

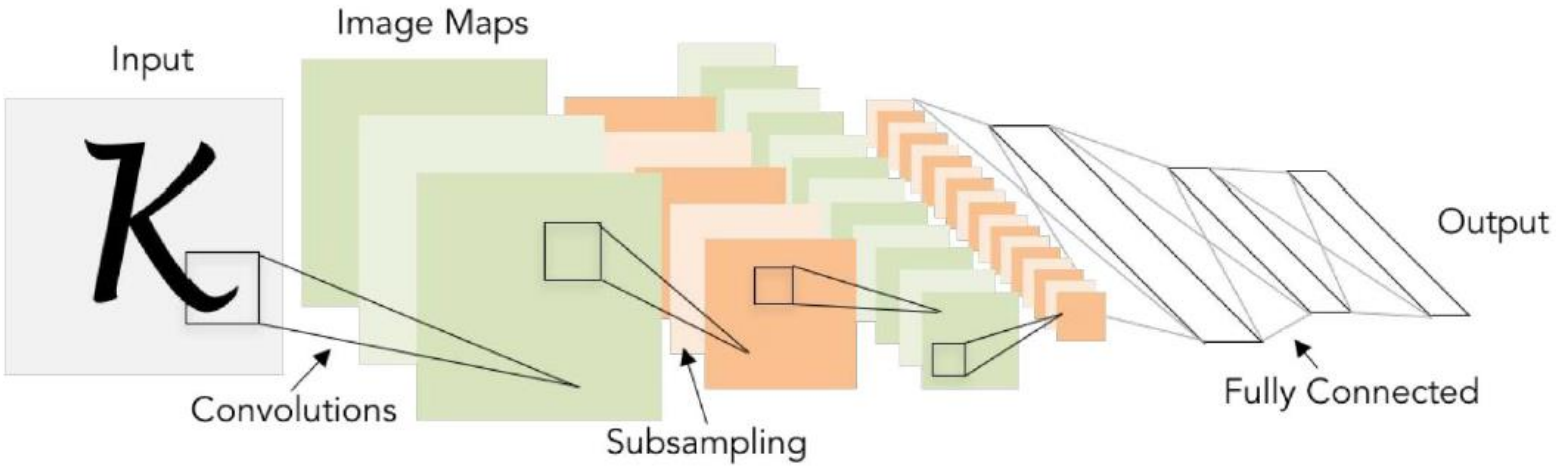
- Problem Setup
  - Input: Image
  - Output: Object class



LeNet-5

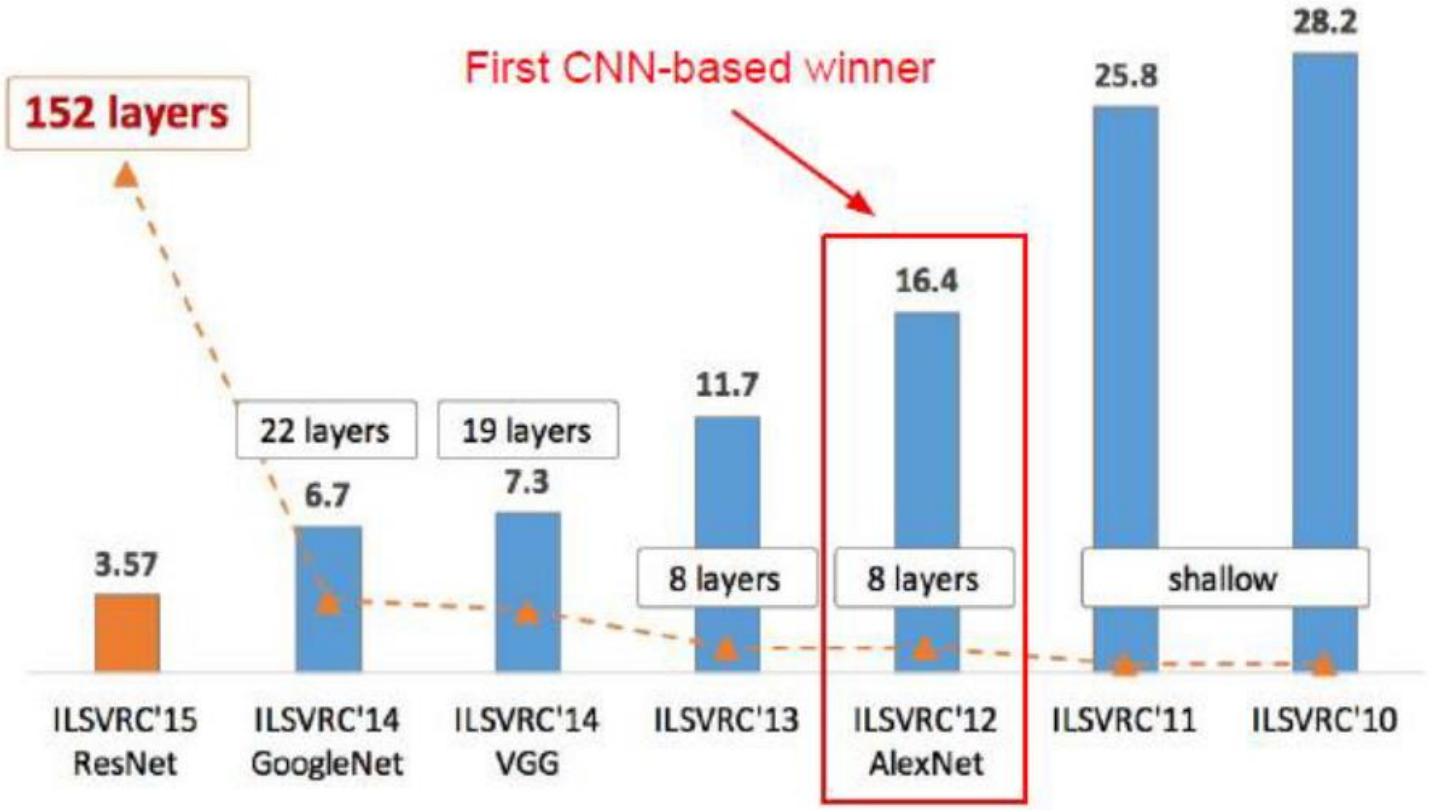
- Handwritten digit recognition

[LeCun et al., 1998]



Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

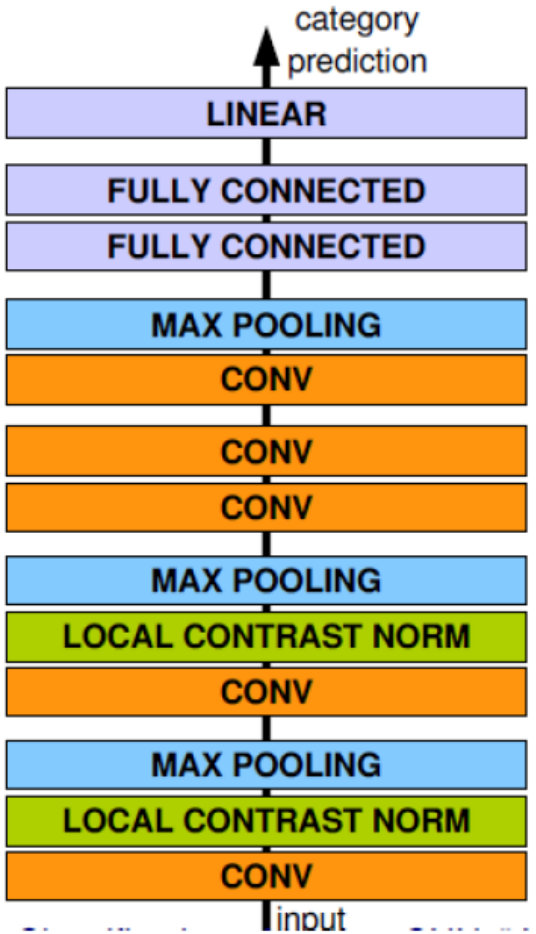
ImageNet (ILSVRC)



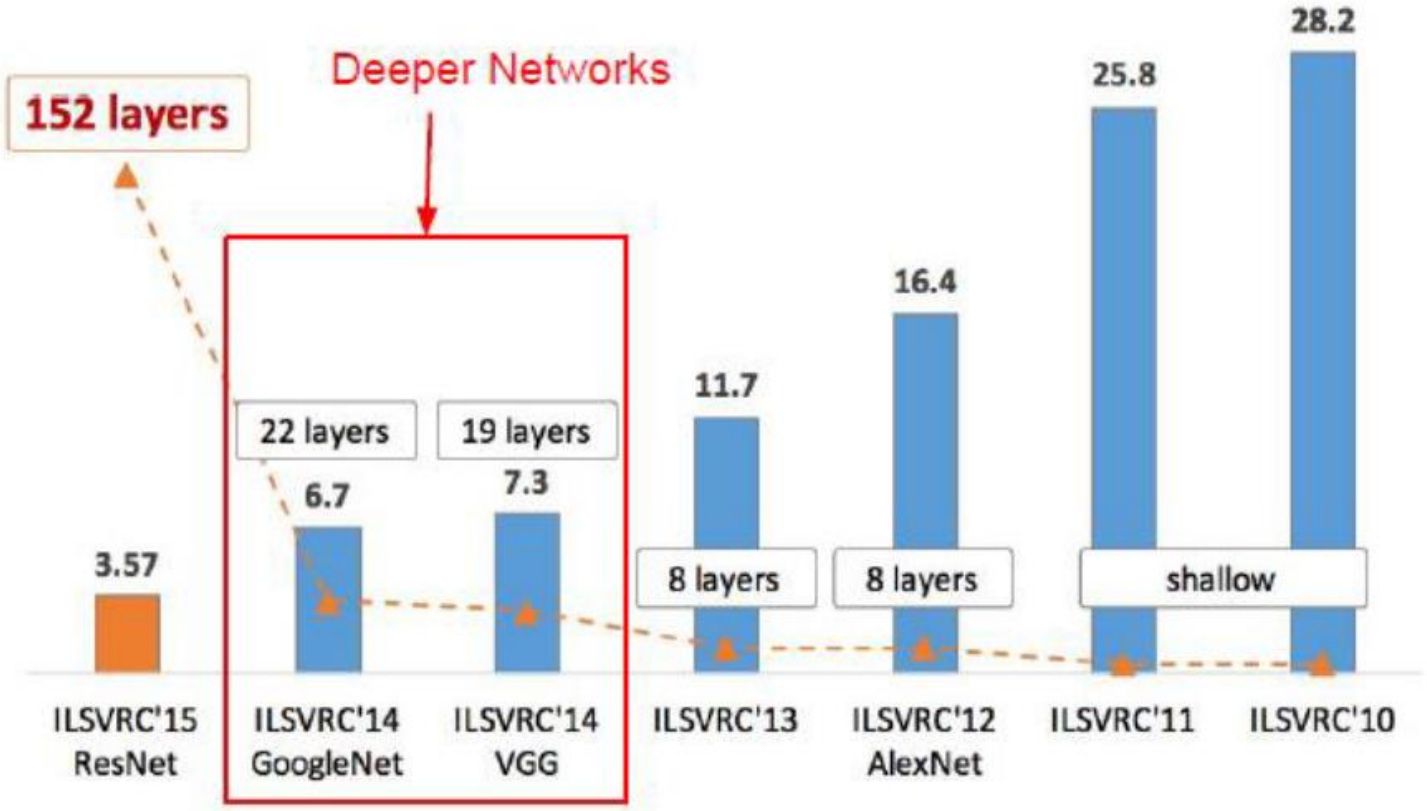
AlexNet

- Deeper network structure
  - More convolution layers
  - Local contrast normalization
  - ReLu instead of Tanh
  - Dropout as regularization

Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8



ImageNet (ILSVRC)



VGGNet

Case Study: VGGNet

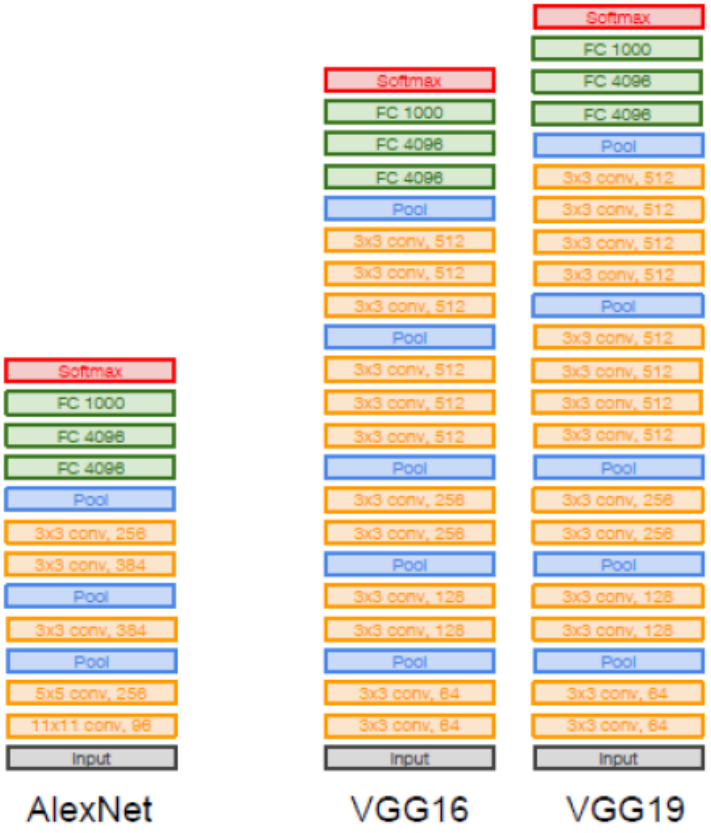
[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14



VGGNet

- Parameters

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = **102,760,448**
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

Note:

Most memory is in early CONV

Most params are in late FC

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

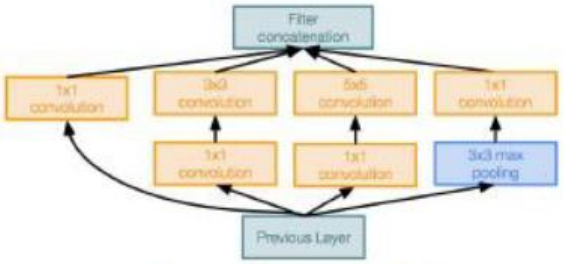
GoogLeNet

Case Study: GoogLeNet

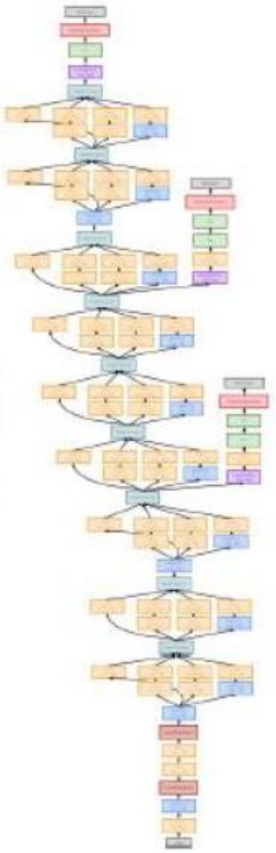
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient "Inception" module
- No FC layers
- Only 5 million parameters!
  12x less than AlexNet
- ILSVRC'14 classification winner
  (6.7% top 5 error)

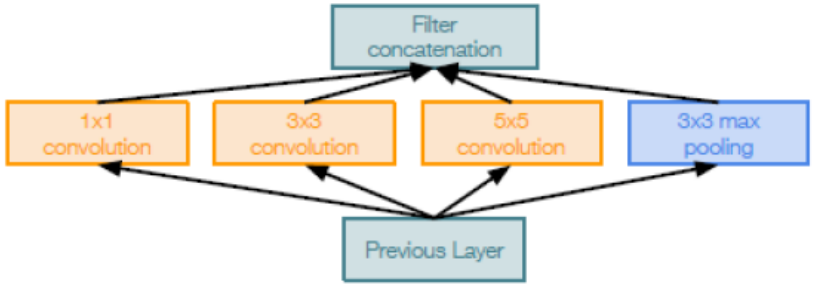


Inception module



GoogLeNet

- Inception Module



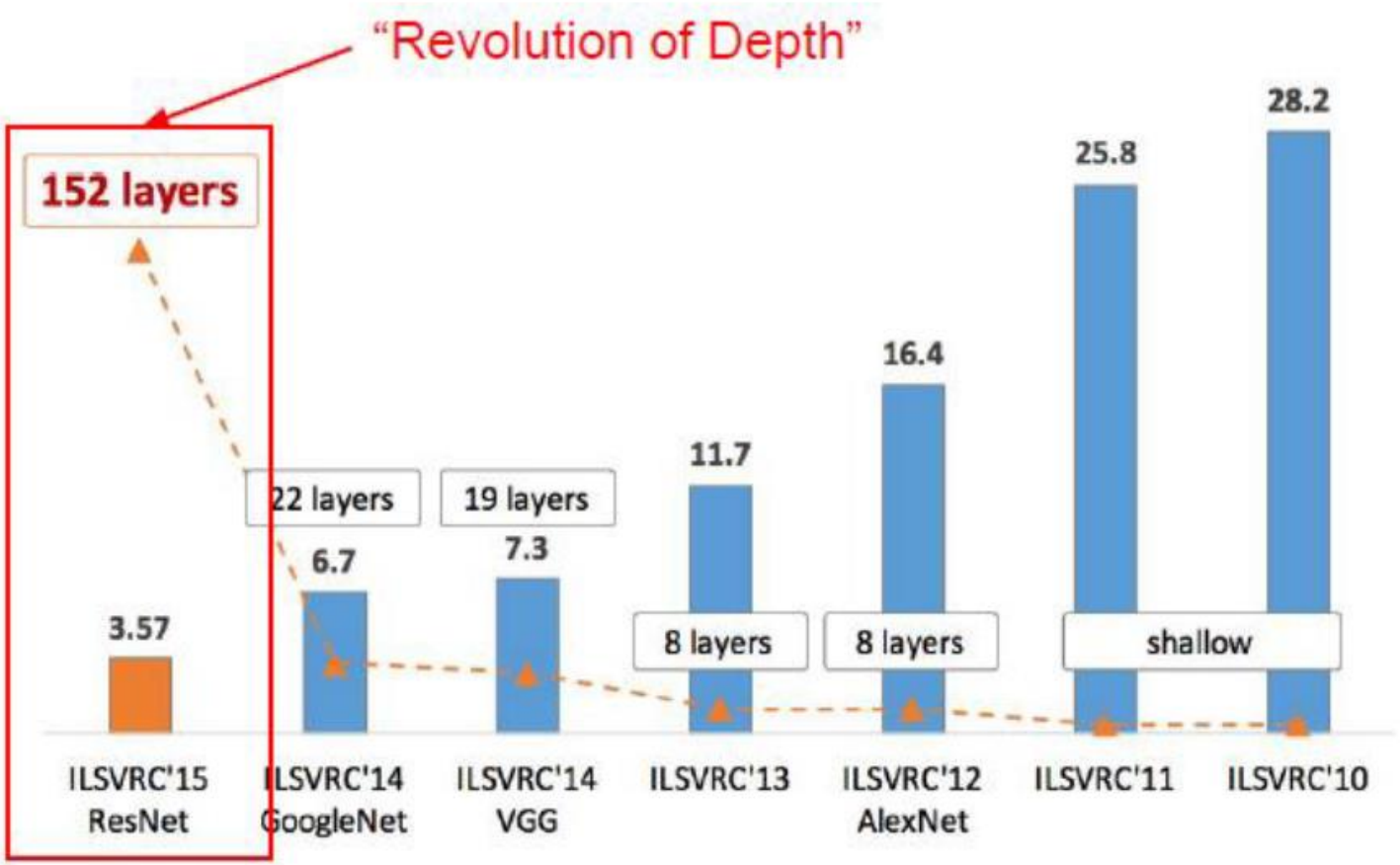
Naive Inception module

Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together depth-wise

ImageNet (ILSVRC)



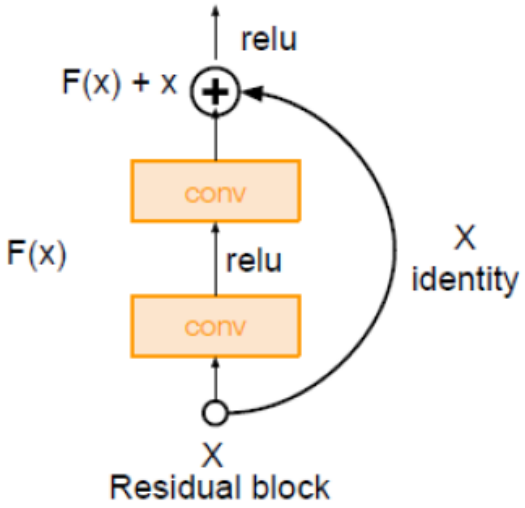
ResNet

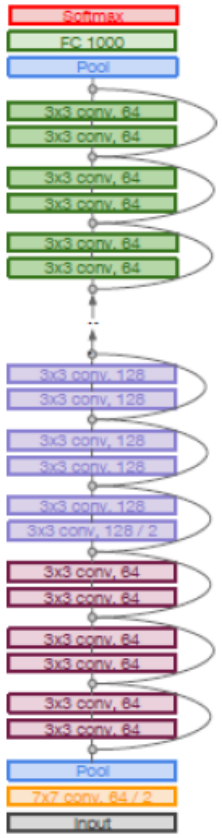
Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

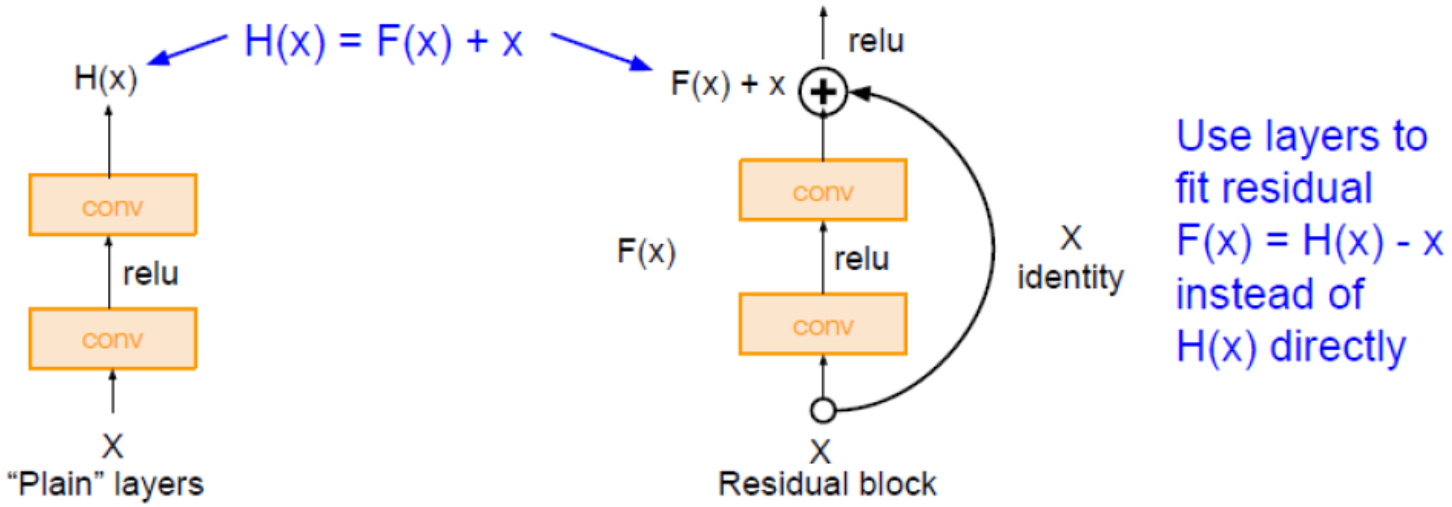
- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!





ResNet

- Solution:
  - Use network layers to fit a residual mapping

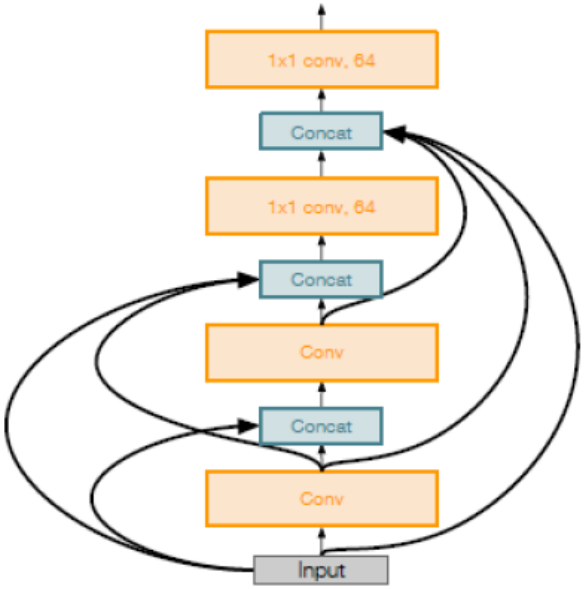


DenseNet

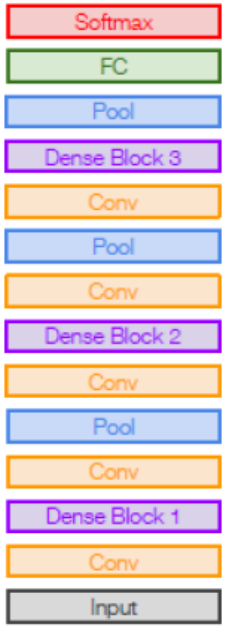
Densely Connected Convolutional Networks

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse

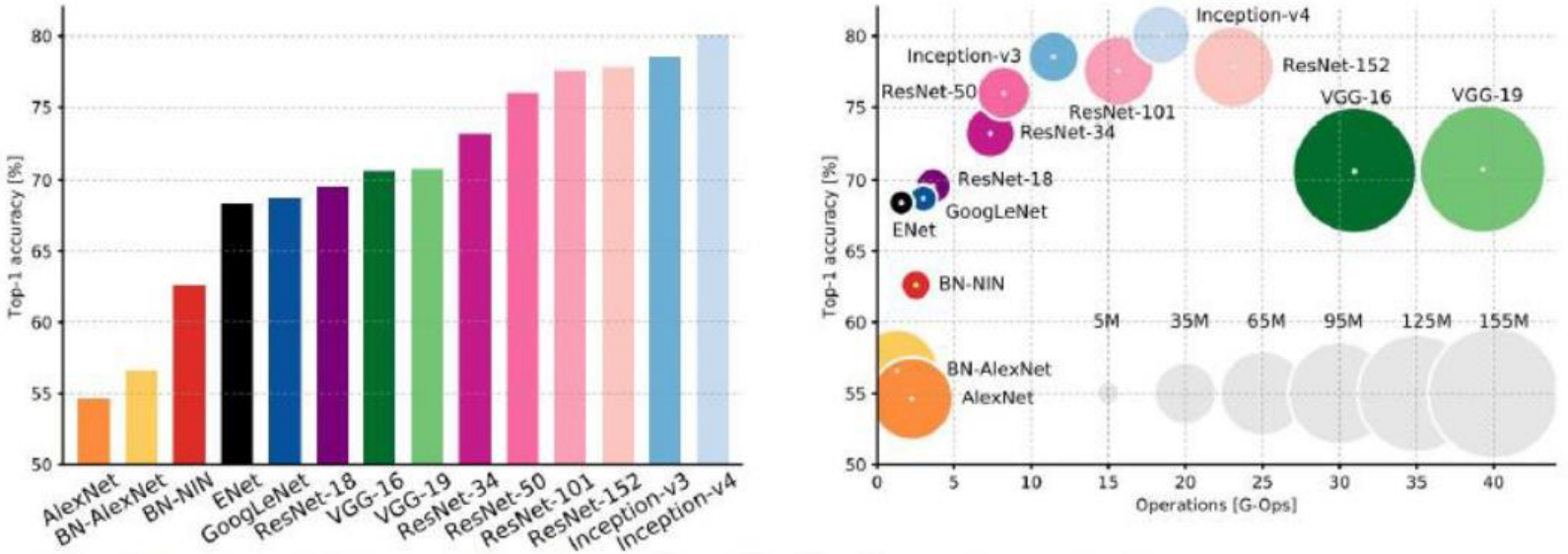


Dense Block



Model Complexity

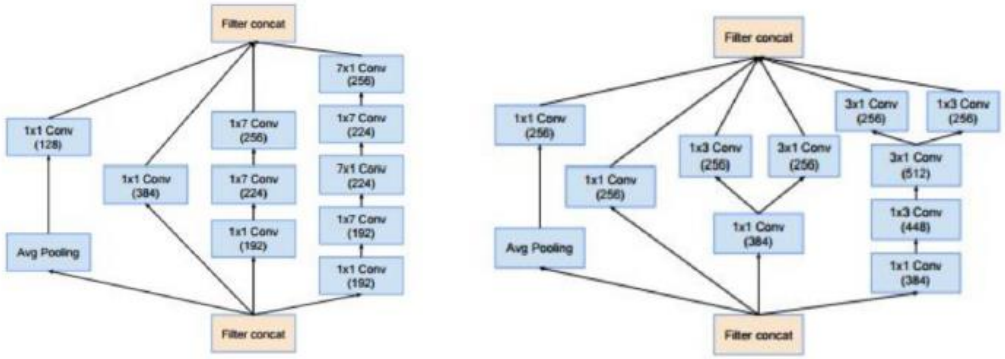
Comparing complexity...



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Network Architecture (NA)

- Problems with network architecture
  - Designing NA is hard
  - Lots of human efforts go into tuning them
  - Not a lot of intuition into how to design them well
  - Can we learn good architectures automatically?



Two layers from the famous Inception V4 computer vision model.
Szegedy et al, 2017

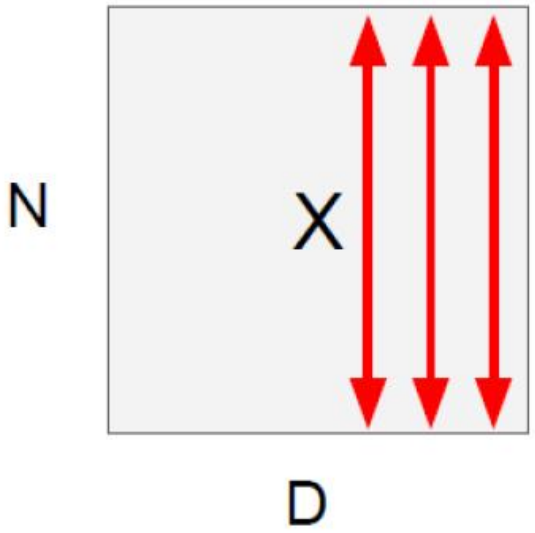
Convolutional Neural Networks-CNN

- Overview
- Architecture
- Optimization and Regularization
- Practice in Python
- Applications

Batch Normalization

- Normalize the inputs to a layer:

"you want unit gaussian activations?
just make them so."



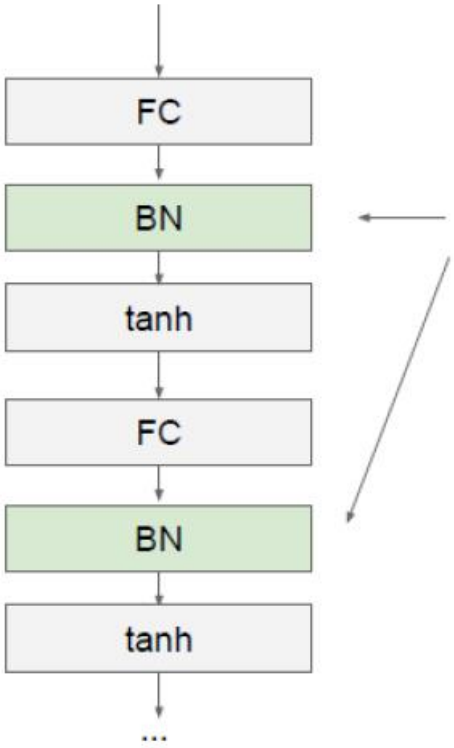
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

- Layer details



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

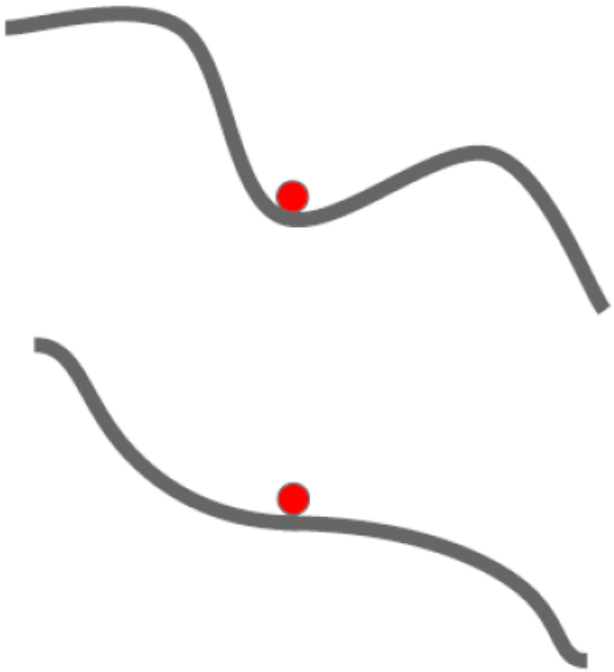
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Optimization

- Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck



Optimization

- SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

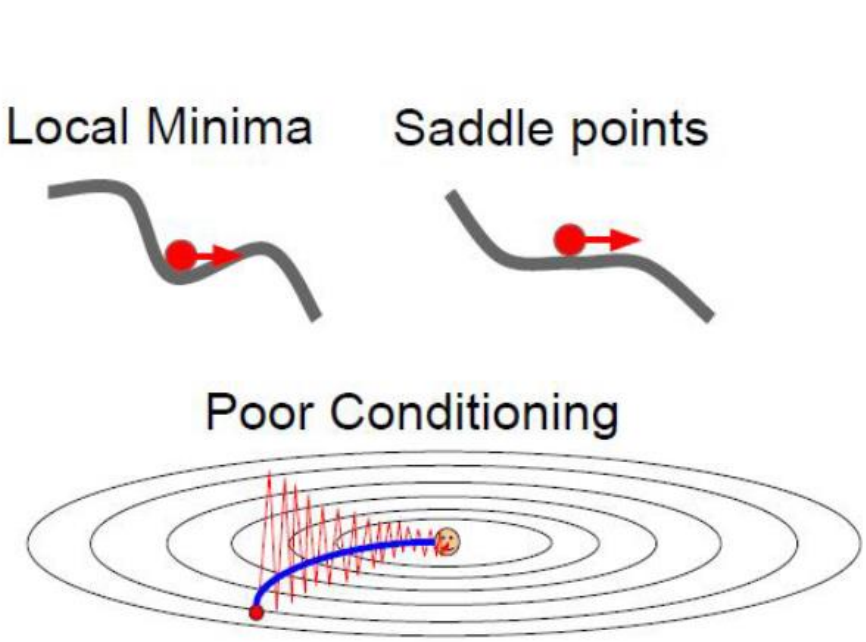
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

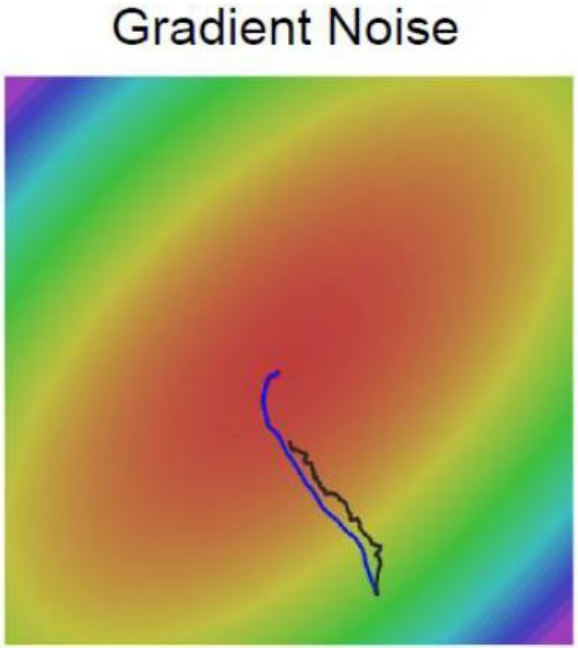
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Optimization

- SGD + Momentum
  - Momentum sometimes helps a lot, and almost never hurts





Optimization

- AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

Optimization

- RMSProp

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

Optimization

- Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

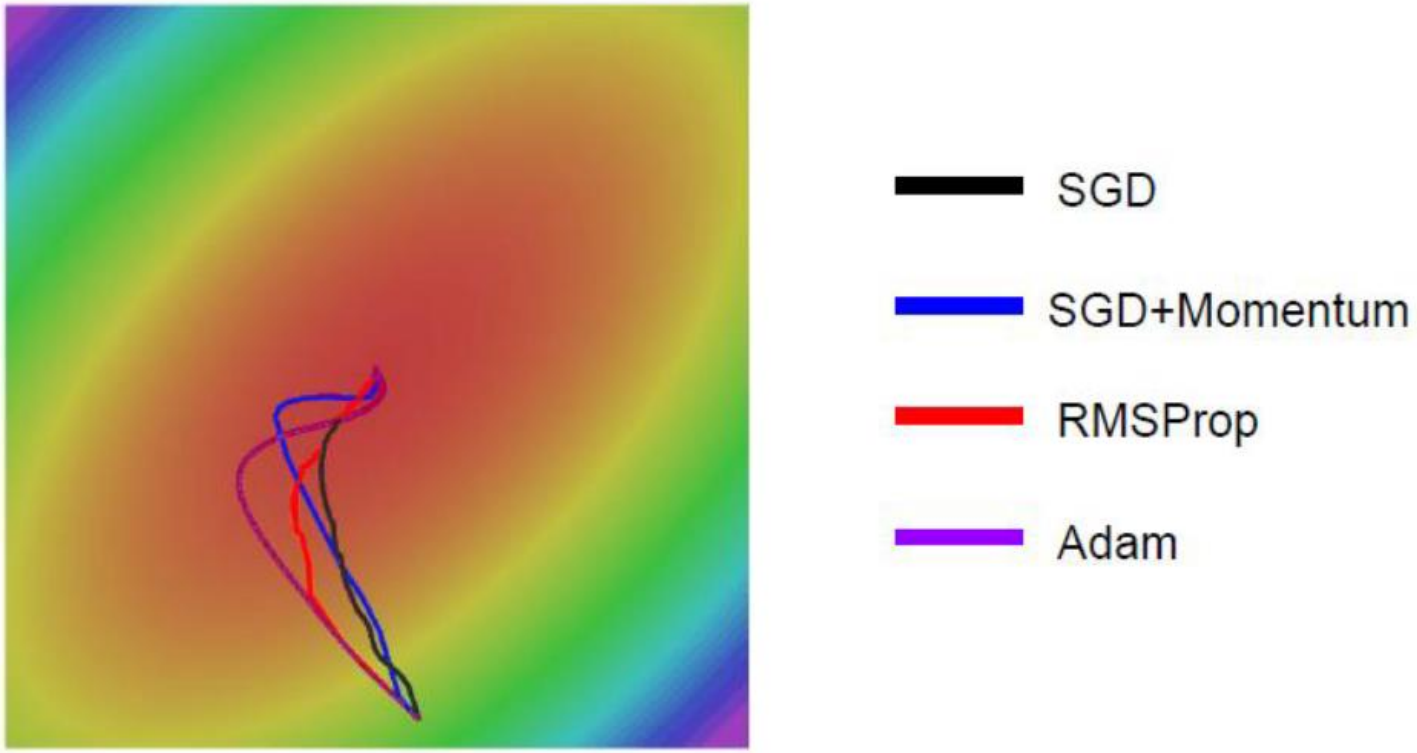
Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

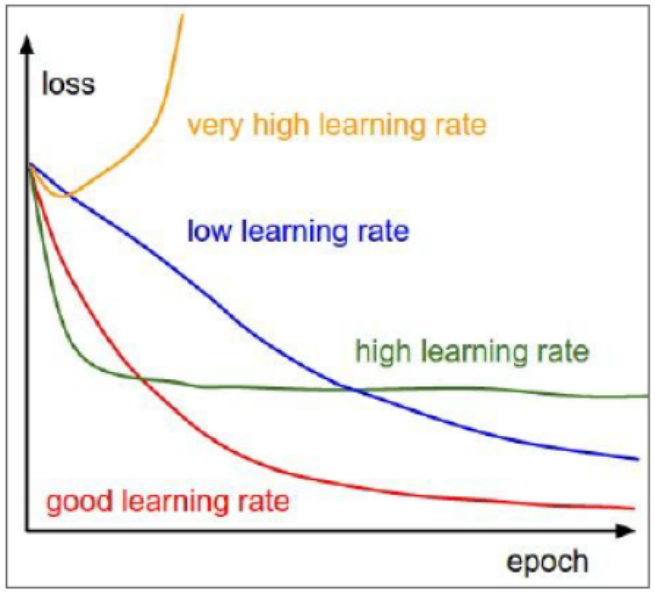
Optimization

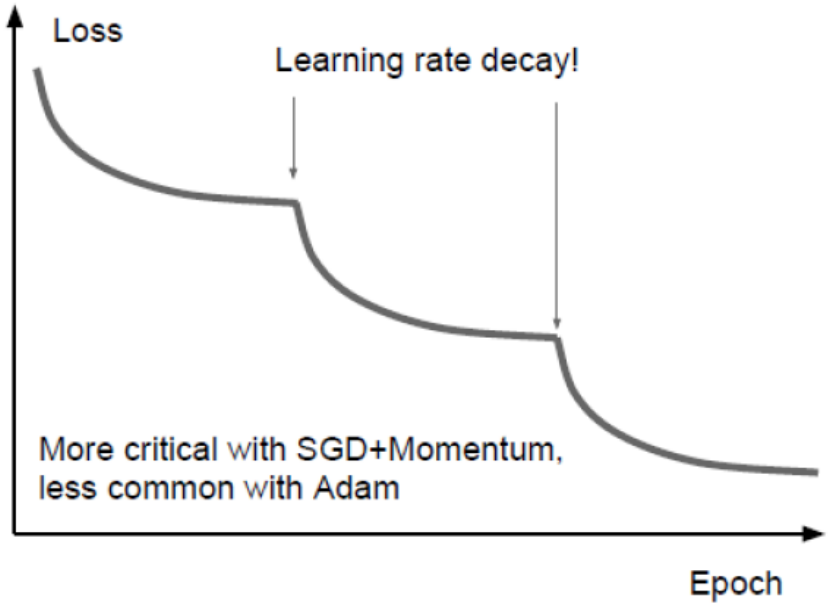
- Adam (full form)



Optimization

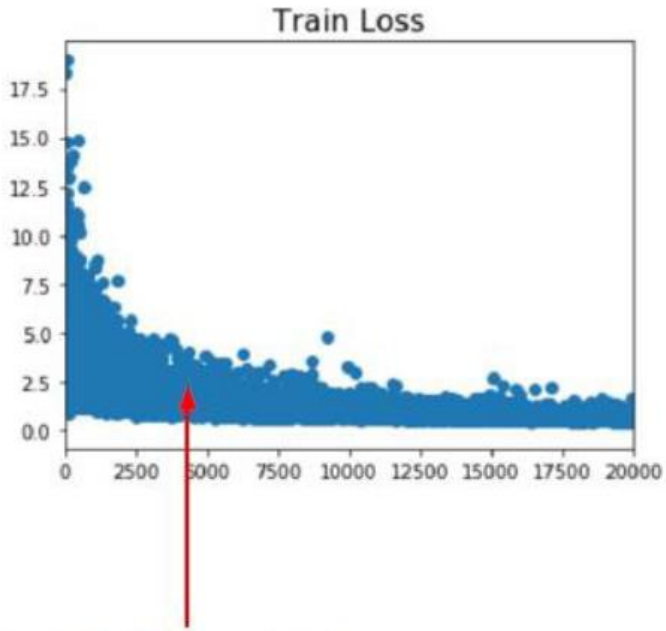
- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter



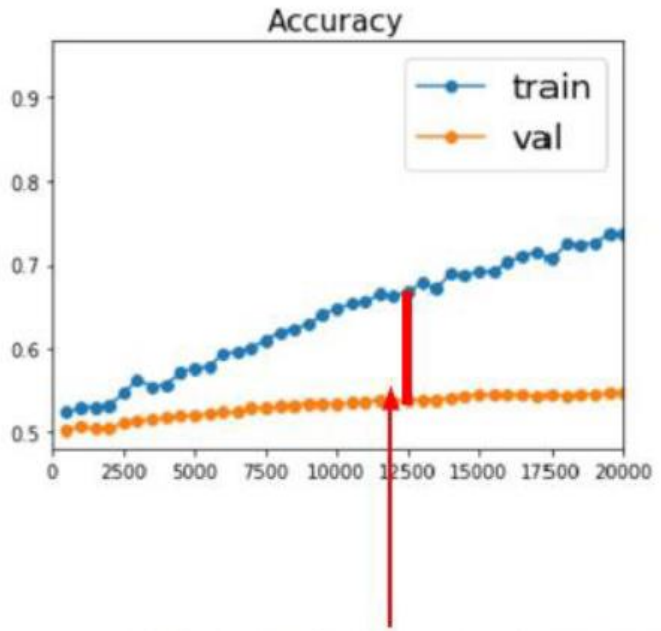


Beyond Training Error

- How do we generalize to unseen data?
  - Well studied but still poorly understood



Better optimization algorithms help reduce training loss



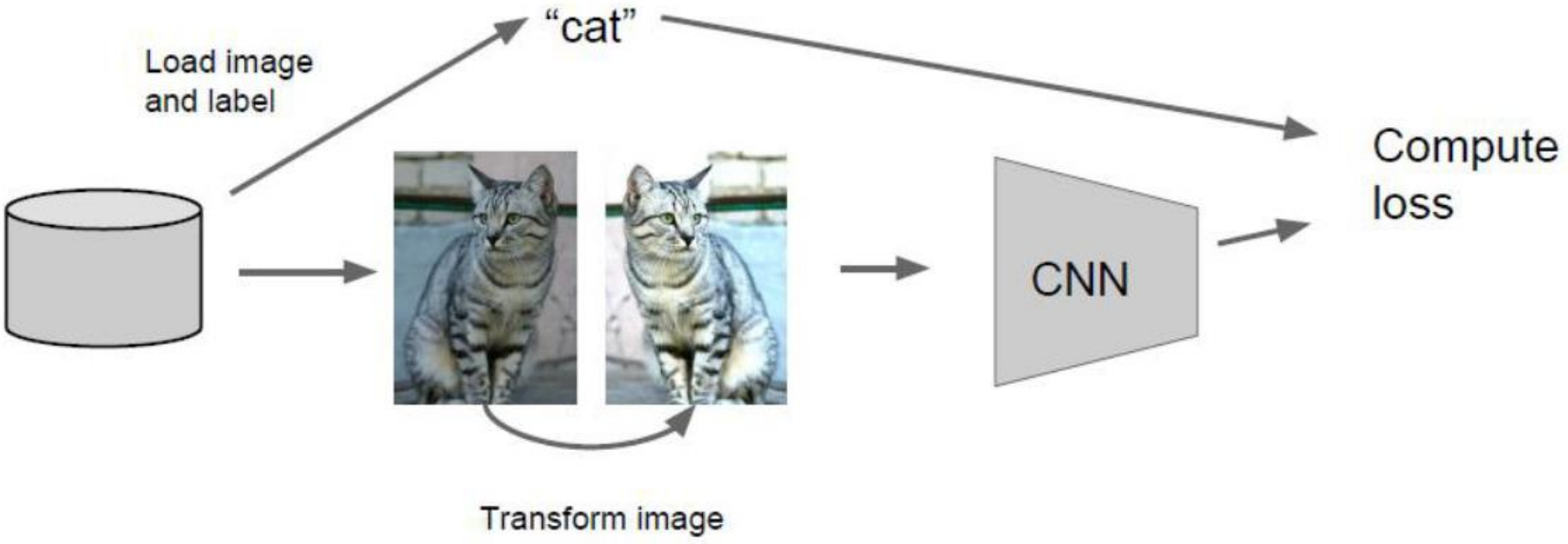
But we really care about error on new data - how to reduce the gap?

Outline

- Regularization in CNN training
  - Data Augmentation
  - Stochastic Regularization
  - Model Ensembles
  - Transfer Learning

Data Augmentation

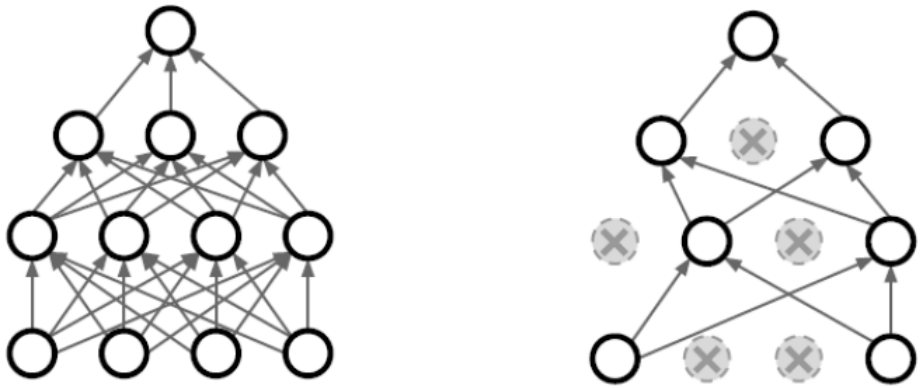
- Create more data for regularization



Examples: translation, horizontal or vertical, flip, rotation, noise…

Stochastic Regularization

- For a network to overfit, its computations need to be really precise. This suggests regularizing them by injecting noise into the computations, a strategy known as stochastic regularization.
- Dropout is a stochastic regularizer which randomly deactivates a subset of the units



Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

Dropout

- Operations

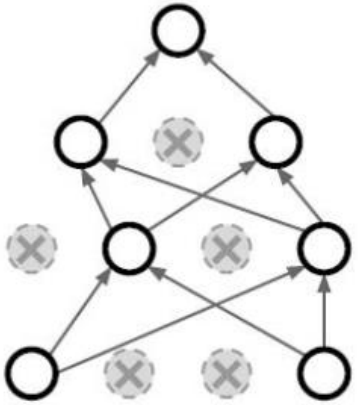
$$h_i = m_i \cdot \phi(z_i),$$

where m_i is a Bernoulli random variable, independent for each hidden unit.

Regularization: Dropout

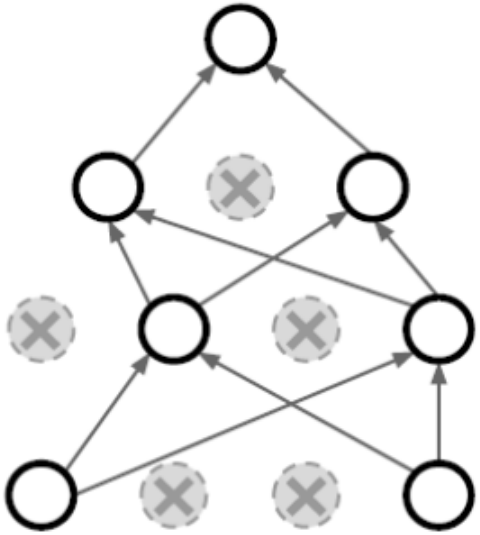
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout

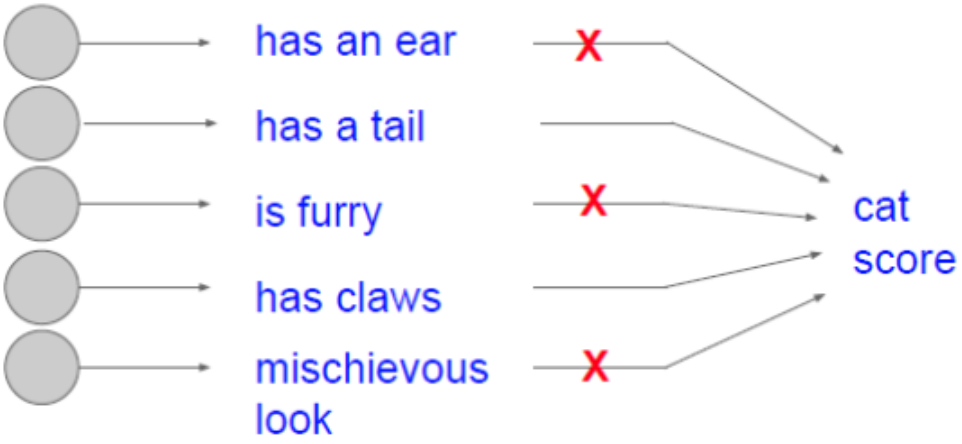


Understanding Dropout

How can this possibly be a good idea?

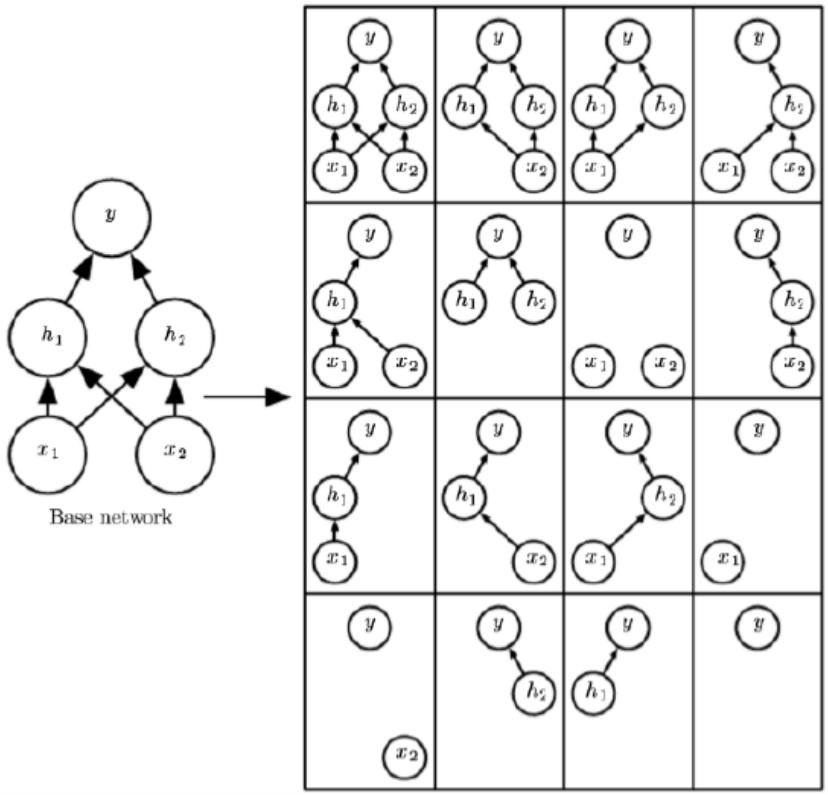


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Understanding Dropout

- Dropout can be seen as training an ensemble of 2^D different architectures with shared weights (where D is the number of units):



— Goodfellow et al., *Deep Learning*

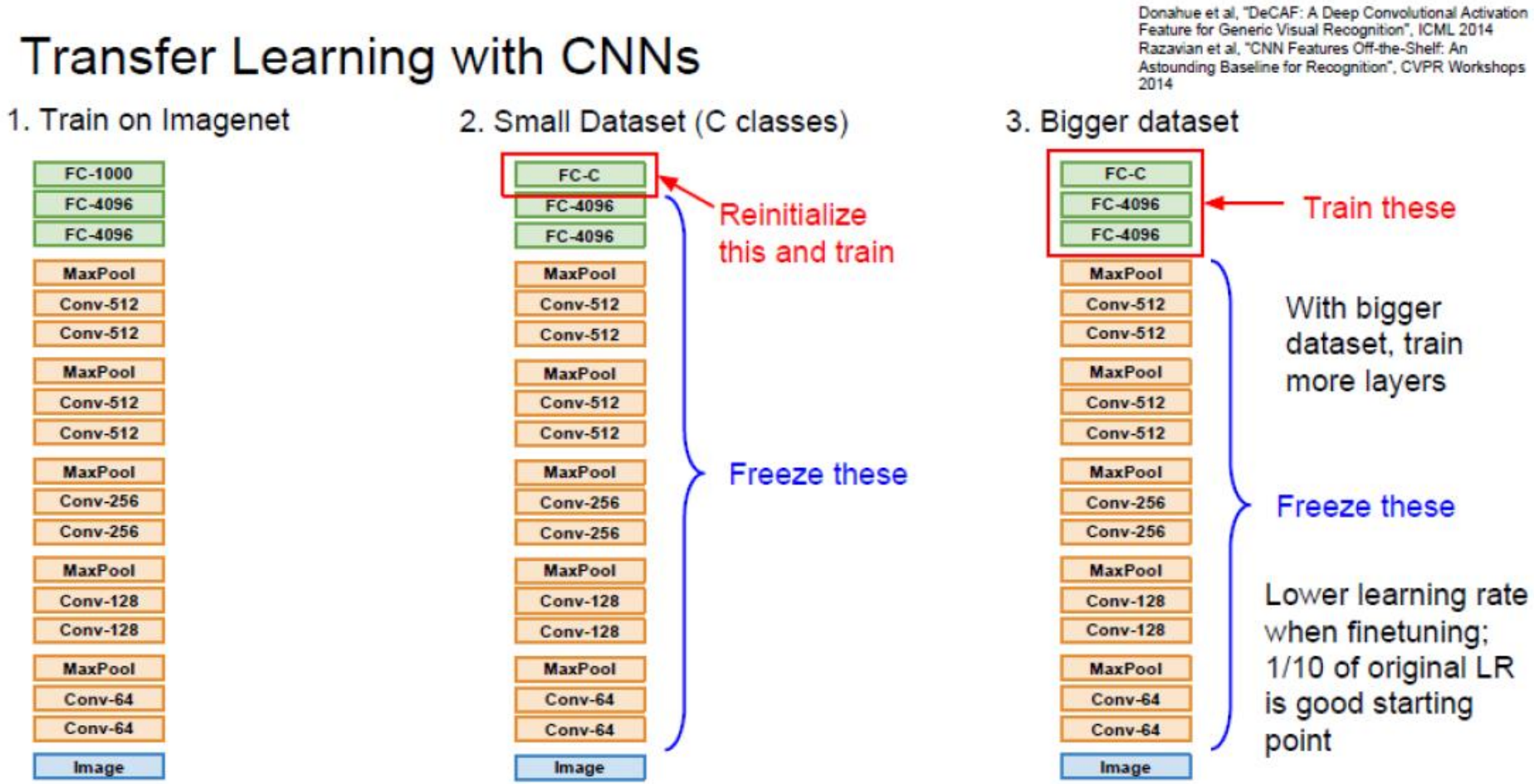
Model Ensembles

- Tips and Tricks

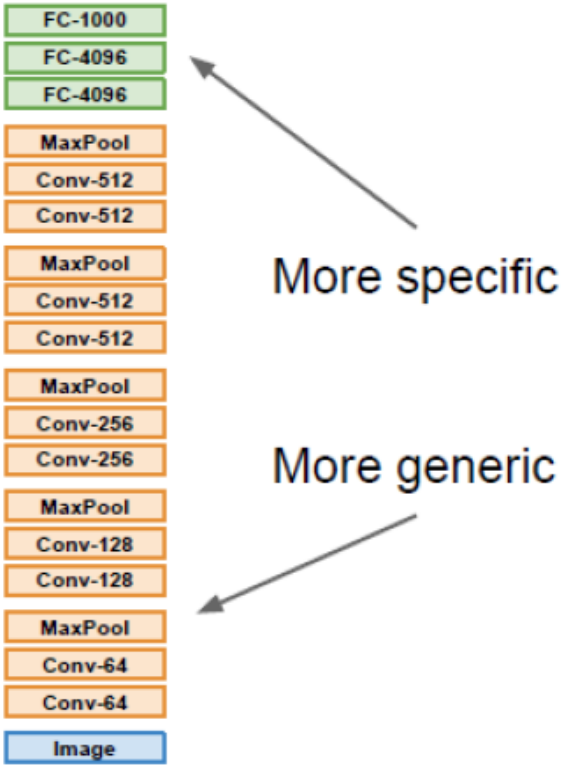
  Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

Transfer Learning



Transfer Learning



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Convolutional Neural Networks-CNN

- Overview
- Architecture
- Optimization and Regularization
- Practice in Python
- Applications

CNN Classification – MINST

Based on Numpy and Pytorch

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
print("PyTorch Version: ",torch.__version__)
```

CNN Classification – MINST

Define a simple CNN for MINST

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1) # 28 * 28 -> (28+1-5) 24 * 24
        self.conv2 = nn.Conv2d(20, 50, 5, 1) # 20 * 20
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        # x: 1 * 28 * 28
        x = F.relu(self.conv1(x)) # 20 * 24 * 24
        x = F.max_pool2d(x,2,2) # 12 * 12
        x = F.relu(self.conv2(x)) # 8 * 8
        x = F.max_pool2d(x,2,2) # 4 *4
        x = x.view(-1, 4*4*50) # reshape (5 * 2 * 10), view(5, 20) -> (5 * 20)
        x = F.relu(self.fc1(x))
        x= self.fc2(x)
        # return x
        return F.log_softmax(x, dim=1) # log probability
```

CNN Classification – MINST

Training and testing

```
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        pred = model(data) # batch_size * 10
        loss = F.nll_loss(pred, target)

        # SGD
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if idx % 100 == 0:
            print("Train Epoch: {}, iteration: {}, Loss: {}".format(
                epoch, idx, loss.item()))
```

CNN Classification – MINST

Training and testing

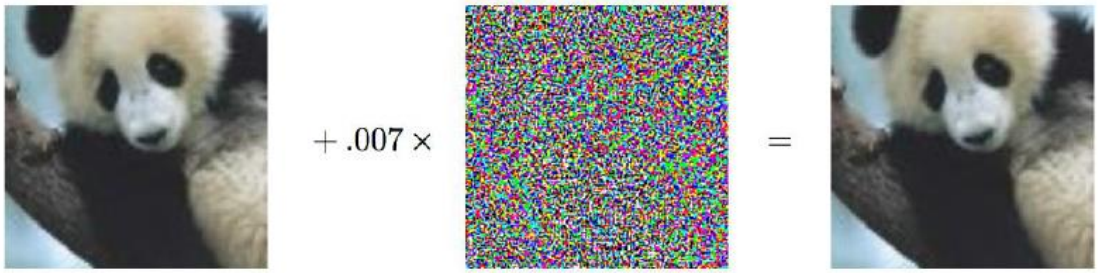
```
Train Epoch: 0, iteration: 0, Loss: 2.283817768096924
Train Epoch: 0, iteration: 100, Loss: 0.6110288500785828
Train Epoch: 0, iteration: 200, Loss: 0.18155980110168457
Train Epoch: 0, iteration: 300, Loss: 0.31043028831481934
Train Epoch: 0, iteration: 400, Loss: 0.518582284450531
Train Epoch: 0, iteration: 500, Loss: 0.1202855259180069
Train Epoch: 0, iteration: 600, Loss: 0.0989612340927124
Train Epoch: 0, iteration: 700, Loss: 0.09637182205915451
Train Epoch: 0, iteration: 800, Loss: 0.13470694422721863
Train Epoch: 0, iteration: 900, Loss: 0.06548292934894562
Train Epoch: 0, iteration: 1000, Loss: 0.03107370436191559
Train Epoch: 0, iteration: 1100, Loss: 0.03948028385639191
Train Epoch: 0, iteration: 1200, Loss: 0.09810394793748856
Train Epoch: 0, iteration: 1300, Loss: 0.15199752151966095
Train Epoch: 0, iteration: 1400, Loss: 0.016710489988327026
Train Epoch: 0, iteration: 1500, Loss: 0.005827277898788452
Train Epoch: 0, iteration: 1600, Loss: 0.0754864513874054
Train Epoch: 0, iteration: 1700, Loss: 0.012112855911254883
Train Epoch: 0, iteration: 1800, Loss: 0.03425520658493042
Test loss: 0.07333157858848571, Accuracy: 97.71
```

Convolutional Neural Networks-CNN

- Overview
- Architecture
- Optimization and Regularization
- Practice in Python
- Applications

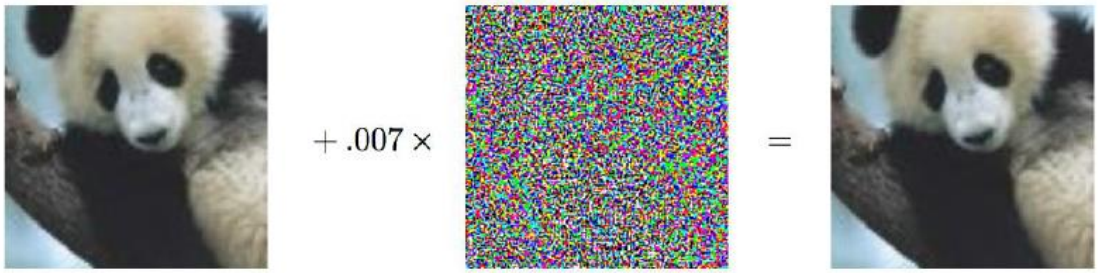
Adversarial Examples

- Surprising findings: adversarial inputs
  - Inputs optimized to fool an algorithm
- Given an image for one category (e.g., “cat”), compute the input gradient to maximize the network’s output for a different category (e.g., “dog”)
  - Perturb the image very slightly in this direction and the network will change its prediction
  - Fast gradient sign method: take the sign of the entries in the gradient



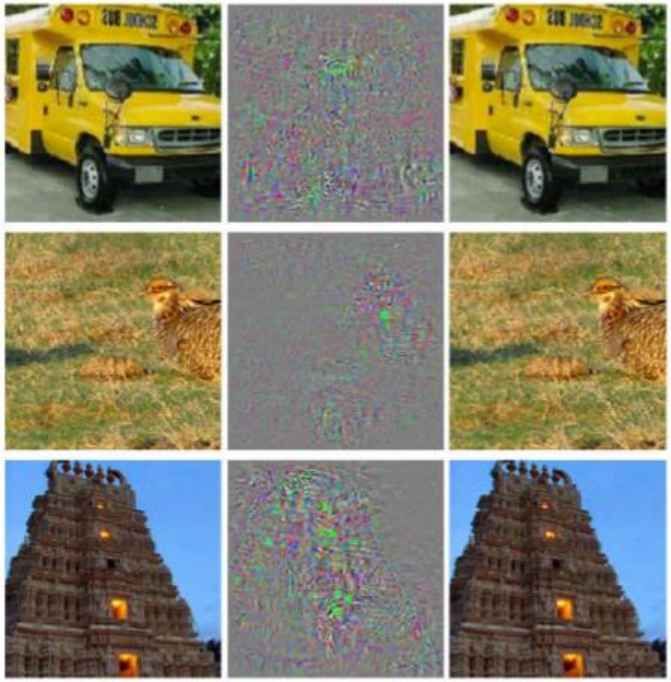
Adversarial Examples

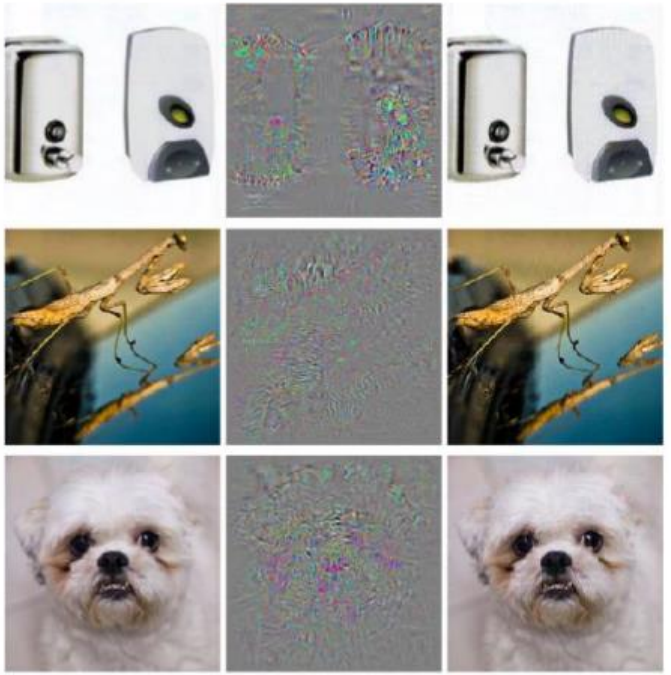
- Surprising findings: adversarial inputs
  - Inputs optimized to fool an algorithm
- Given an image for one category (e.g., “cat”), compute the input gradient to maximize the network’s output for a different category (e.g., “dog”)
  - Perturb the image very slightly in this direction and the network will change its prediction
  - Fast gradient sign method: take the sign of the entries in the gradient



Adversarial Examples

- The following adversarial examples are misclassified as ostriches (Middle = perturbation x 10)





Neural Style Transfer

- Problem setup

Content Image

This image is licensed under CC-BY 3.0

+

Style Image

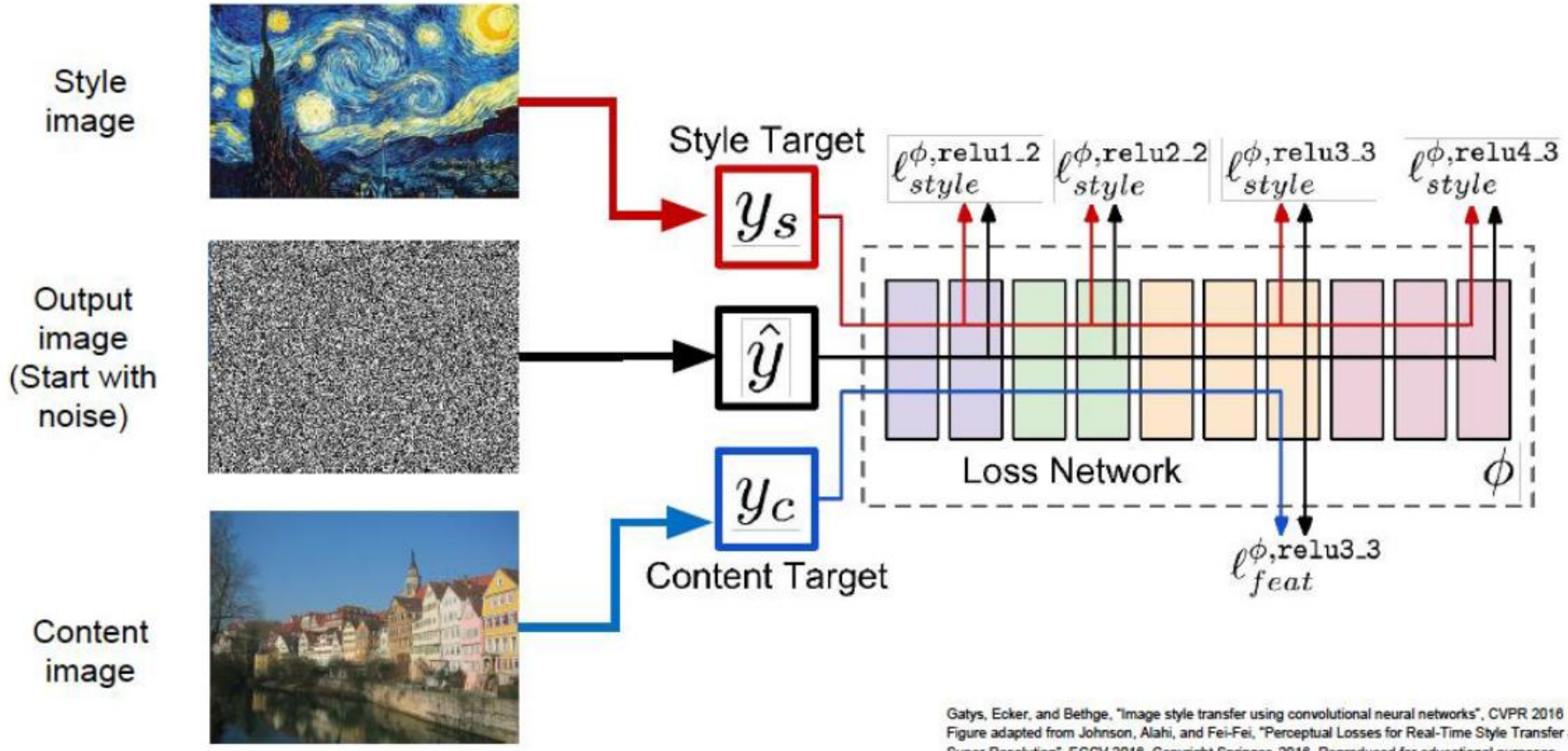
Starry Night by Van Gogh is in the public domain

=

Style Transfer!

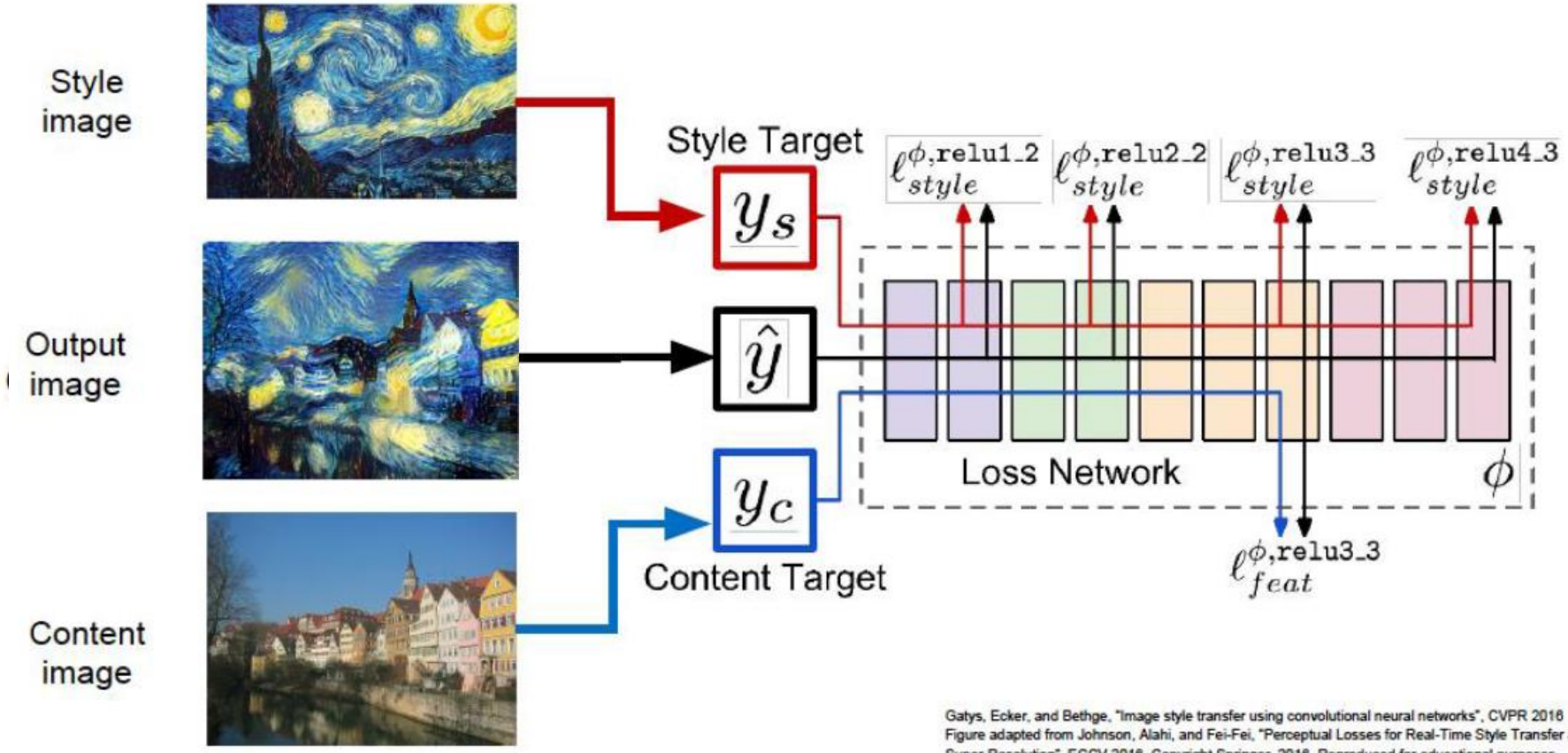
This image copyright Justin Johnson, 2015. Reproduced with permission.

Neural Style Transfer



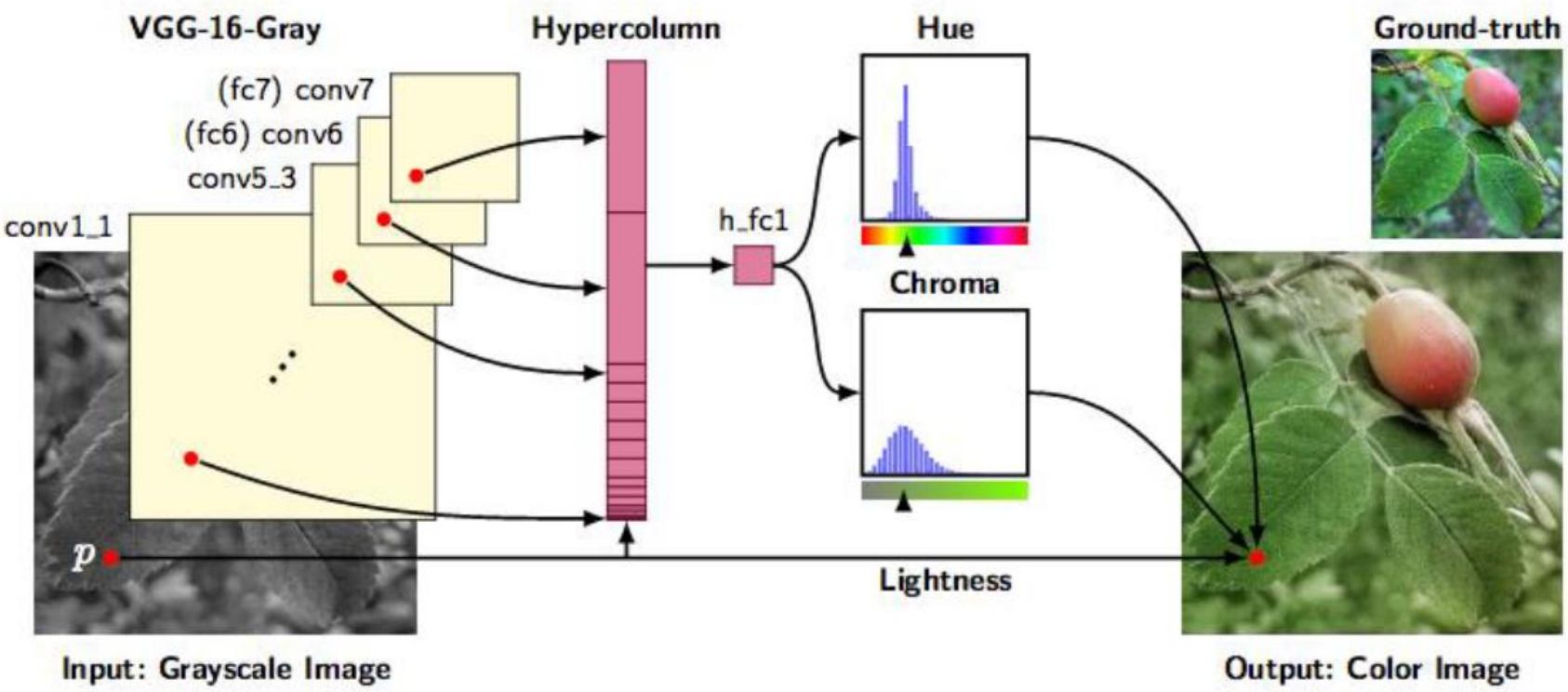
Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.

Neural Style Transfer



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.

Automatic Colorization



http://people.cs.uchicago.edu/~larsson/colorization/

Further Reading

- Alexnet: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
- VGG: https://arxiv.org/pdf/1409.1556.pdf
- Resnet: https://arxiv.org/pdf/1512.03385.pdf
- Inception-Resnet v4: https://arxiv.org/pdf/1602.07261.pdf
- MobileNet: https://arxiv.org/pdf/1704.04861.pdf
- YOLOv3: https://pjreddie.com/media/files/papers/YOLOv3.pdf
- Original Style Transfer: https://arxiv.org/pdf/1508.06576.pdf
- Dropout: http://jmlr.org/papers/v15/srivastava14a.html