Deep Learning for Visual Computing

Deep Image Classification

Christopher Pramerdorfer

Computer Vision Lab, TU Wien

Image from twitter/mechanicmind_ai, created using midjourney

This Week in AI
Segment Anything

Image from segment-anything

Topics

Deep learning for image classification

Convolutional neural networks

- Convolutional layers
- Pooling layers
- Classification backends

Basic classification architectures

Motivation

At this point we know

- That we must tackle large-scale CV problems with ML
- How to pose image classification as an ML task
- How to extract low-level features from images
- What neural networks are and how to train them

This is the traditional image classification pipeline

- Some low-level feature extractor (e.g. HoG)
- Some form of dimensionality reduction (e.g. PCA)
- Some generic classification model (e.g. MLP)

Motivation

This pipeline performs poorly on large-scale problems

- Can only extract indiscriminative low-level features
- CIFAR-10 test accuracy only $\approx 60\%$ (HoG + MLP)

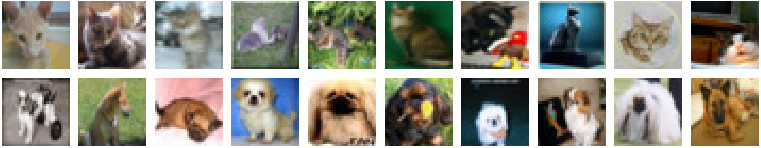


Image from cs.toronto.edu

Motivation

We cannot design reliable high-level feature extractors

But we can try to learn them

- Approach is called representation learning

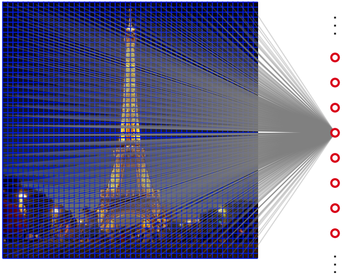
In theory we can do so using MLPs

- Use image vectors as inputs
- Hidden layer(s) extract features
- Output layer acts as linear classifier

Motivation

This usually does not work well though

- $\dim(\boldsymbol{\theta})$ increases quickly with image size & model capacity
- MLPs are designed for arbitrary vector inputs, not images
- Complex MLPs are hard to train



Motivation

We clearly need a better neural network architecture

- Optimized for image data
- Fewer parameters per layer (easier to increase depth)

Let us derive such an architecture from MLPs

- By introducing reasonable inductive biases
- Assumptions built into the model or training pipeline
- To improve model performance and/or efficiency

Convolutional Neural Networks
Input Layer

Network should make use of spatial structure of images

- Not sensible to flatten images to vectors

We represent them as 3D tensors $\mathbf{X}$ instead

- Tensors are n-dimensional generalizations of matrices

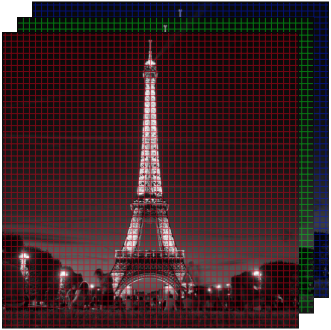
We will use $C \times H \times W$ dimension order

- Same order as PyTorch
- $3 \times 32 \times 32$ for CIFAR-10

Convolutional Neural Networks
Input Layer

Input neurons form $C_0 \times H_0 \times W_0$ grid

Input Image



Input Layer

Spatially close pixels are highly correlated, others are not

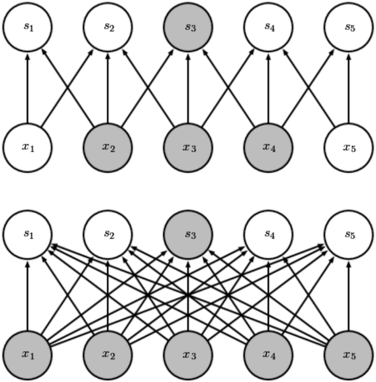
- Nearby pixels likely correspond to same object (part)

A good image feature extraction layer should account for this

- Compute local features from spatially close inputs

We can achieve this by

- Arranging hidden layer neurons in $W_l \times H_l$ grid
- Connecting only spatially close neurons
- Neurons are thus sparsely connected (fewer parameters)



Sparse Connectivity

Dense Connectivity (MLP)

Image adapted from [1]

W_l and H_l depend on input width and height

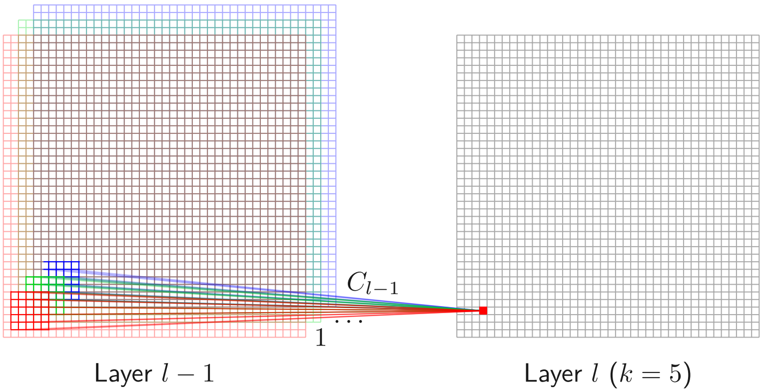
- Usually $W_l = W_{l-1}$ and $H_l = H_{l-1}$ to preserve resolution
- Padding in border regions (replication)

Connectivity k along W and H dimensions

- Configurable but often $k = 3$, that is 3×3

Connectivity along channel dimension is usually C_{l-1}

- Want to make use of all local information



C_{l-1}
$\cdots$
1
Layer $l-1$
Layer l $(k=5)$

Convolutional Neural Networks
Feature Extraction

Extraction should work the same anywhere in input

- We generally don't know where objects will appear
- Due to varying object location and viewpoint

Achieved by letting neurons compute the same operation

- For linear layers this means identical weights and bias
- So $o_h = \mathbf{W}_l \cdot \mathbf{X}_h + b_l$ with $\mathbf{X}_h, \mathbf{W}_l \in \mathbb{R}^{C_{l-1} \times k \times k}$

Neurons in layer l compute $o_h = \mathbf{W}_l \cdot \mathbf{X}_h + b_l$

- $\mathbf{W}_l \cdot \mathbf{X}_h$ is a linear combination (like before)
- $\mathbf{W}_l$ is identical for all neurons in layer

The overall transformation of the layer is thus

- A convolution of the input with kernel $\mathbf{W}_l$
- Followed by an additive bias b_l

Such layers are thus called convolutional layers

- Or conv layers for short

Recall how discrete convolutions work (here $C_{l-1} = 1$)

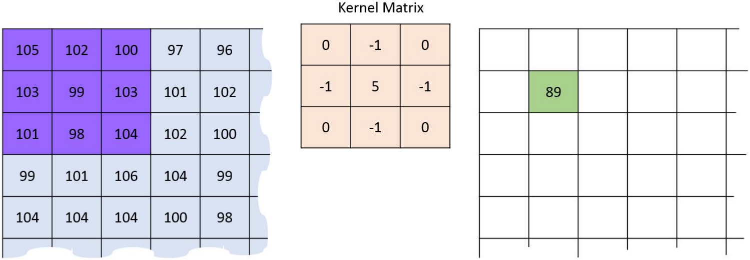


Image Matrix

$$105 * 0 + 102 * -1 + 100 * 0$$
$$+103 * -1 + 99 * 5 + 103 * -1$$
$$+101 * 0 + 98 * -1 + 104 * 0 \; = 89$$

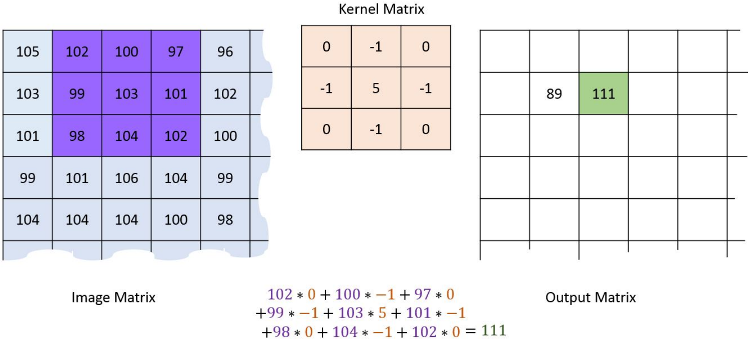
Output Matrix

Image from machinelearninguru.com

Convolutional Neural Networks

Feature Extraction

Recall how discrete convolutions work (here $C_{l-1} = 1$)



$$102*0+100*-1+97*0$$
$$+99*-1+103*5+101*-1$$
$$+98*0+104*-1+102*0=111$$

Image from machinelearninguru.com

Neurons thus "detect" features via $o_h = \mathbf{W}_l \cdot \mathbf{X}_h + b_l$

- Respond to local structures similar to $\mathbf{W}_l$
- Similar to template matching with learned template $\mathbf{W}_l$

Conv layers are thus rather simple feature extractors

- Power comes from stacking such layers
- With activation functions (ReLU) and other layers in between

One issue remains

- Every neuron performs same operation
- So layer can learn to extract only one feature

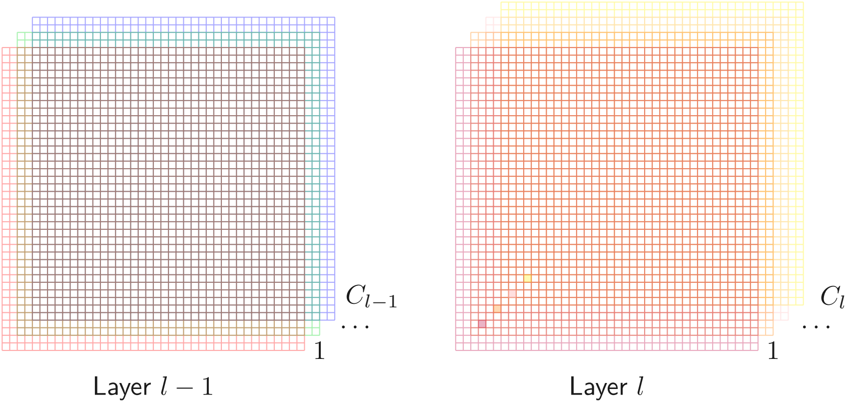
To overcome this problem we replicate the neurons C_l times

- Resulting in a $C_l \times W_l \times H_l$ grid of neurons
- Arranged in C_l feature maps of size $W_l \times H_l$

Layer can thus learn C_l different features

- Only neurons in same feature map (channel) share parameters

Convolutional Neural Networks
Feature Extraction



Convolutional Neural Networks
Feature Extraction

Number of weights $\mathbf{W}_l$ depends only on k, C_{l-1}, C_l

- $k = 3$, $C_{l-1} = 3$, $C_l = 32 \implies 864$ weights
- $k = 3$, $C_{l-1} = 32$, $C_l = 64 \implies 18.5$k weights

Way fewer parameters than with linear layers

- Can stack many conv layers
- Layer l learns to combine layer $l - 1$ features to new ones

Neurons see only small part of previous layer (sparse connectivity)

- But larger input region (receptive field) as depth increases

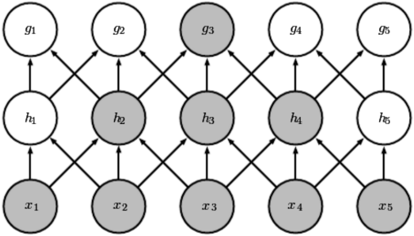


Image from [1]

Convolutional Neural Networks
Feature Extraction

So in networks of conv layers

- Direct connections are sparse
- But receptive field can span most/all of image

Feature extraction approach is thus part-based

- Learn local features (e.g. presence of eye or nose)
- Learn more global features (e.g. presence of face) from those

Hierarhical approach to representation learning

- Divide and conquer

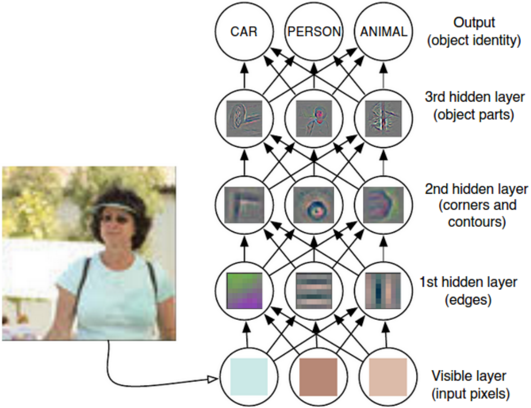

CAR
PERSON
ANIMAL
Output
(object identity)
3rd hidden layer
(object parts)
2nd hidden layer
(corners and
contours)
1st hidden layer
(edges)
Visible layer
(input pixels)

Image from [1]

Conv layers are fundamental deep learning layers

- Part of most network architectures for image analysis

Networks with conv layers are convolutional neural networks

- CNNs or convnets for short

Recall that conv layers

- Retain the input size $W_{l-1} \times H_{l-1}$ (padding)
- Or reduce it only slowly (no padding)

This

- Slows down computations
- Leads to shallow receptive fields

We thus want some form of pooling

- Reduce W_{l-1} and H_{l-1} via local aggregation

Pooling layers are an example

- Process input channels independently
- Aggregate via $\max(\mathbf{X}_h)$ or $\mathrm{mean}(\mathbf{X}_h)$
- Leave C_{l-1} unchanged

2×2 max-pooling layer with stride 2

- $W_l = W_{l-1}/2$, $H_l = H_{l-1}/2$, and $k = 2$
- Output of neuron h is $\max(\mathbf{X}_h)$ with $\mathbf{X}_h \in \mathbb{R}^{2\times 2}$

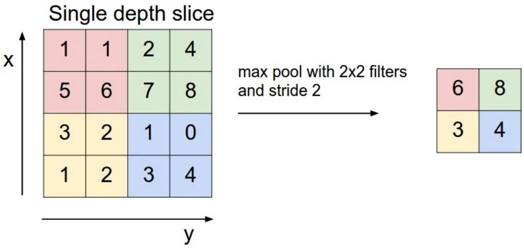
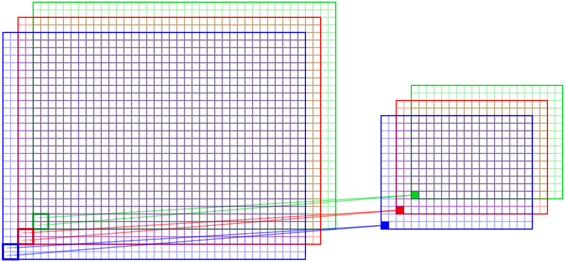


Image from cs231n.github.io

Convolutional Neural Networks
Pooling

Number of neurons reduced by factor 4

- Corresponding efficiency increase
- At the cost of losing spatial resolution



Can also use strided convolutions

- Conv layer with e.g. stride 2
- Popular replacement for max-pooling layers

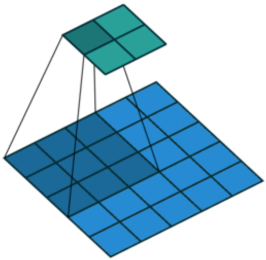


Image from github.com

How much pooling do we need?

- Depends on task
- Usually W_l and H_l end up < 10

Modern classification architectures pool down to 1×1 (!)

- Via a final global average-pooling layer
- Pooling using $\text{mean}(\cdot)$ and $k = W_l = H_l$

Conv and pooling layers produce 3D tensors $(C_l \times W_l \times H_l)$

For classification we convert to vectors $\mathbf{x}_l$

- Flatten input tensor like we did earlier with images

Allows us to connect linear layers, resulting in

- A linear or non-linear (MLP) classifier
- That processes vectors of learned features

Convolutional Neural Networks
Classification Backends

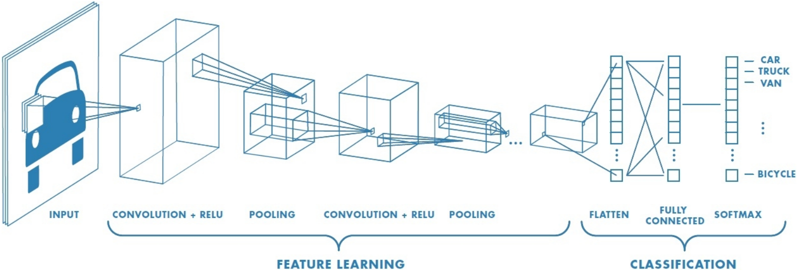


Image from floydhub.com

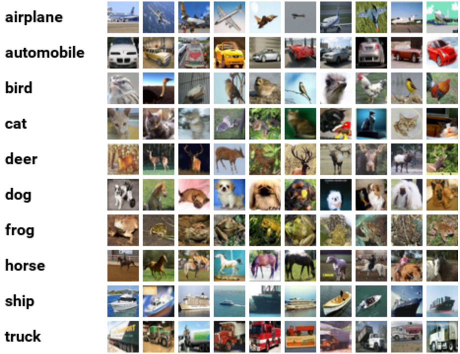
Modern architectures use linear classifiers

- So the learned features are so powerful
- That the simplest classifier is sufficient (!)

We finally have task-specific high-level features

- Reason why deep learning is so powerful

Convolutional Neural Networks
CIFAR-10 Demo



Demo

This concludes the basic layer types and purposes

- Conv + ReLU layers for feature extraction
- Pooling (or strided conv) layers for dimensionality reduction
- Linear layers for classification

The question is how to arrange these layers properly

- Following slides introduce a simple recipe
- More advanced designs will covered in next lectures

Use square images, $H_0 = W_0 = R$

- Resize images such that smaller side has size R
- Then extract a center crop of size $R \times R$

R should be divisible by 2 many times

- Avoid problems during pooling

Start with small R

- And test if increasing R makes sense
- $R = 224$ is popular for classification
- But much smaller R might be sufficient

Get R below 100 quickly via aggressive pooling

- To improve efficiency
- $R = 224$: conv with $k = 7, s = 2 \Rightarrow 2 \times 2$ max-pooling

Use conv $\Rightarrow$ conv $\Rightarrow$ pooling blocks

- Conv layers with $k = 3$, stride 1, padding, and ReLUs
- Pooling layers with 2×2 max-pooling with stride 2

Start with 32 or 64 feature maps

- Increase by factor 2 in each subsequent block
- Up to a maximum of 512 feature maps

Stack blocks until $R \leq 7$

- Usually means 4 or 5 such blocks

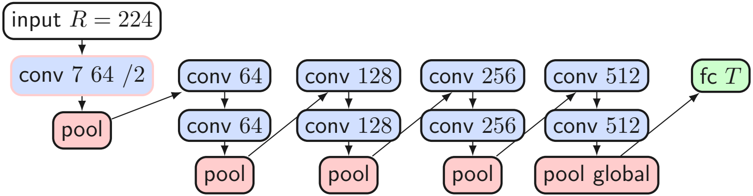
Finish with linear classifier

- Global average pooling
- Followed by linear layer with T neurons

This recipe is a good starting point

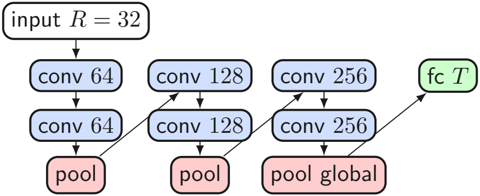
- Decent performance on many datasets (try yourself)
- Tune some hyperparameters depending on time budget

Example result with $R = 224$



Example result for for our cat vs. dog problem ($R = 32$)

- Try different variants yourselves



Let's implement the above network in PyTorch

- Based on subnetworks for components
- One of many options (could just stack layers)

First import the required modules:

```
import torch
import torch.nn as nn
```

Networks always process samples in batches

- So input tensor is 4D
- Batch size is 1 for single images

```
x = torch.zeros(  # dummy input
  4,     # batch size
  3,     # number of input channels
  224,   # imput height
  224    # input width
)
```

Frontend first

```
def frontend(nin, nout):
  return nn.Sequential(
    nn.Conv2d(nin, nout, kernel_size=7, padding=3, stride=2),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=2, stride=2)
  )

f = frontend(3, 64)
y = f(x)  # size of y is [4, 64, 56, 56]
```

Basic building blocks

```
def conv3(nin, nout):
  return nn.Sequential(
    nn.Conv2d(nin, nout, kernel_size=3, padding=1),
    nn.ReLU(inplace=True)
  )

def block(nin, nout=None, pool=True):
  nout = nout if nout else nin * 2
  return nn.Sequential(
    conv3(nin, nout),
    conv3(nout, nout),
    nn.MaxPool2d(kernel_size=2, stride=2) if pool else nn.Identity()
  )
```

Deep Image Classification
PyTorch Implementation

Complete network

```
net = nn.Sequential(
  frontend(3, 64),
  block(64, 64),
  block(64),
  block(128),
  block(256, pool=False),
  nn.AvgPool2d(kernel_size=7),
  nn.Flatten(),
  nn.Linear(512, 10)  # assuming 10 classes
)

y = net(x)  # size of y is [4, 10]
```

Bibliography

[1] Goodfellow et al. Deep Learning. 2016.