Convolutional Neural Networks

- normalize inputs so images are in same range (e.g. 8 bit)
  ↳ mapping of colors, (camera settings, ...)

Train batch based:

1. clear gradients of optimized variables
2. forward pass, compute predicted outputs by passing inputs to model
3. calculate loss
4. Backward pass: compute gradients of the loss w/ respect to our params
5. perform optimization step.
6. update average training loss

When do you stop training?

1. Train set
2. Validation set
3. Test set

} train: update model weights

→ test: check how well the model generalizes

↑ check performance

↓ no backprop

MLP vs CNN

- MLP: only FCL
- MLP: 2D into is lost
- too many parameters

CNN
- sparsely connected
- matrix input
- → MLP at the end

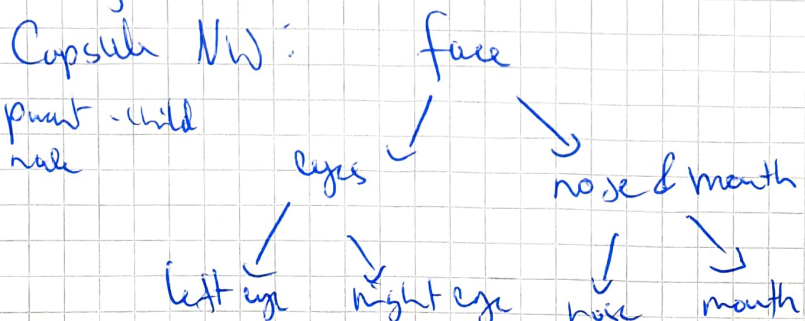
Filters

- learned by the Network to detect edges
- detect ~~faces~~ more complex shapes, the deeper the Network
- the filter Dimension depends on image depth
- move filter ~~above~~ over image
- multiple filters are used, stack them like image colorspaces
- Locally Connected, parameter sharing
- Stride => how do we slide image ?
- Padding -> for edges to not lose information
- Pooling: Reduces Dimensionality
  - ↳ MaxPool (takes max. value of Window)
  - ↳ also has stride
  - ↳ no parameters! -> Dropout doesn't make sense
  - ↳ avg-pool, min-pool, ...
  - ↳ Information is lost (Spatial)

Alternative to pools: Capsule Networks:

-> extract spatial information between parts/objects

Capsule NW:



part-child rule

Capsules

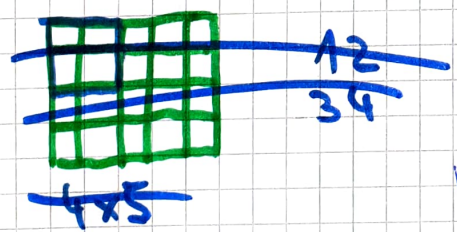
- Collection of nodes, contain info about a specific part & it's properties
- each capsule outputs a vector w/ some magnitude & orientation

→ vector allows to build up a parse tree to recognize whole objects consisting of smaller parts
↳ Magnitude should be high, no matter the orientation

→ Dynamic Routing is used to find best connections from one layer of capsules to the next → even good when objects overlap

Pre-processing

- Images need to be resized



32.10

130 × 130 input

Weights per Filter:
$F^2 \cdot D\_in$ (# K filters)

Output size:

$$(W - F + 2P)/S + 1$$

W: Width, F: Filter Size, P: Padding

(4) Augmentation

- Scale invariant
- Rotation invariant
- Big Pooling → position not as relevant anymore

⇒ Add augmented images
  ↳ Translation
  ↳ Rotation
  ↳ fight Overfitting

→ torchvision transforms library

→ AlexNet → Dropout & ReLU

→ VGG, simple, elegant 3×3 Conv, 2×2 Pool

→ ResNet, shortcut connections
  ↳ vanishing Gradient Problem

→ Visualizing Feature Maps
  ↳ Edges
    ↳ stripes
      ↳ Objects

} Deep Dream

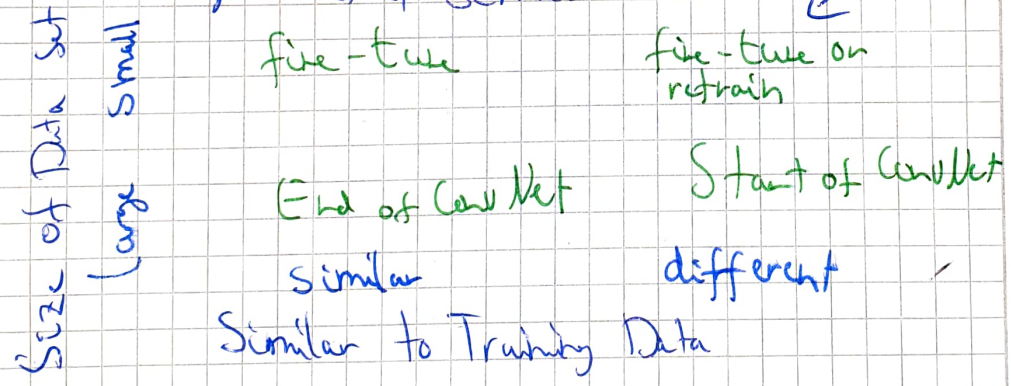
Transfer Learning

→ pre trained Networks
→ transfer it to other task
→ save computation
→ last filters are most specific
↳ Only those need to be trained

Depends on:
- size of data set
- similarity of data

↳ cases:
large data, different data
small data, different data
small data, similar data
large data, similar data

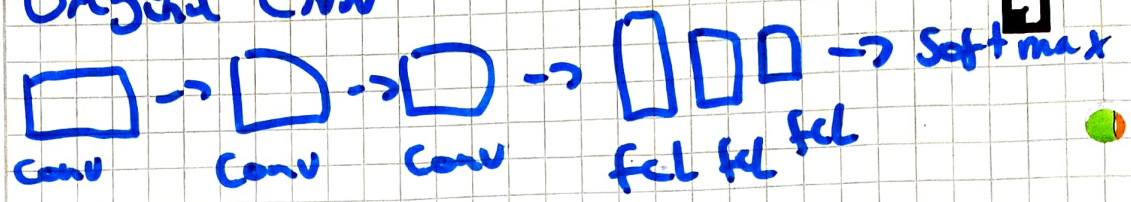
Guide for TL



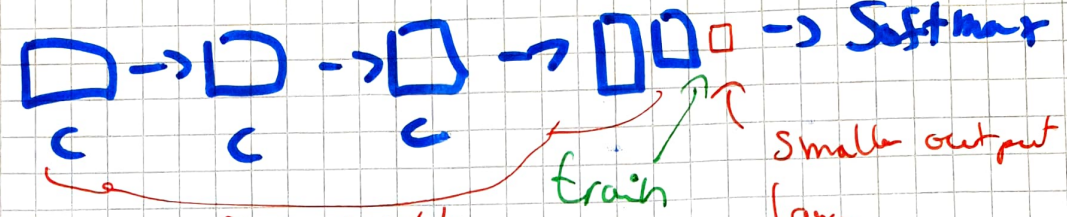
→ Network might need to be tweaked
(output layers)

6

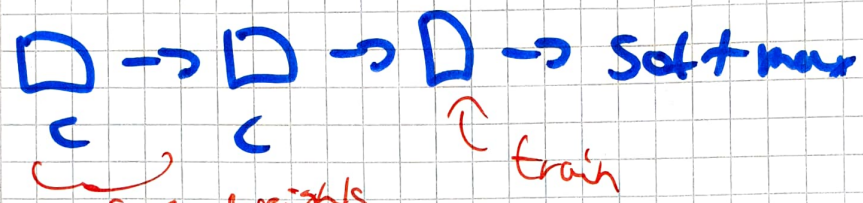
Original CNN



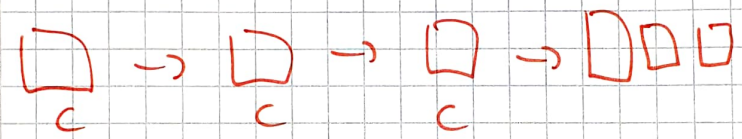
Case small Data set, similar Data



Case small Data set, different Data

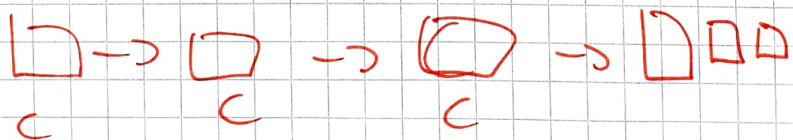


Case large Data, similar Data



retrain entire Network -> pretrained weights as a start in conv layers

Case large Data, different Data



fine-tune or retrain entire Network

-> initialize (with) pre-trained weights might be good

Freeze weights:

```
for param in vgg16.features.parameters():
    param.requires_grad = False
```

Training loop w/ validation

```
valid_loss_min = np.Inf
for epoch in range(1, n_epochs):

    train_loss = 0.0
    valid_loss = 0.0
    for batch_i, (data, target) in enumerate(train_loader):
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        optimizer.zero_grad()
        output = vgg16(data)
        loss = criterion(output, labels)   # target
        loss.backward()
        optimizer.step()
        train_loss += (loss.item() * data.size(0)

    # validation
    for data, target in valid_loader:
        output = model(data)
        loss = criterion(output, target)
        valid_loss += loss.item * data.size(0)

    train_loss = train_loss / len(train_loader.dataset)
    valid_loss = valid_loss / len(valid_loader.dataset)
    if valid_loss <= valid_loss_min:
        valid_loss_min = valid_loss
        torch.save(model.state_dict(), 'model.pt')
```

Save best model

8 Weight initialization

Constant Weight

↳ to close to zero (nn.init)

→ all zero weights:

→ not accurate scores

→ CEL, backpropagation is hard to do, sources of error difficult to tell (less than 10%)

Instead: random uniform distribution

→ Initialize each weight uniform

→ create function to access weights & change them

Good range: → $[-y, y]$ when

$y = 1/\sqrt{n}$ n = number of classes

Normal distribution

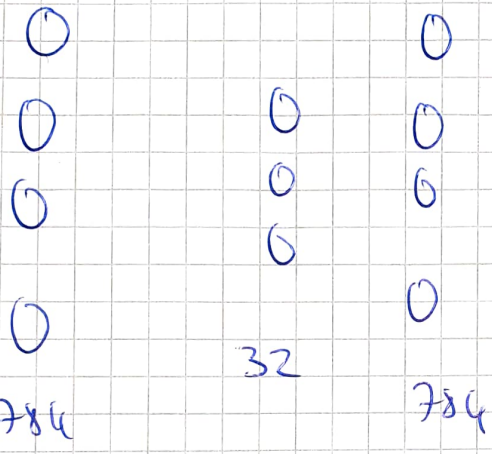
→ equality distribution, Bell curve

→ more values closer to zero

Autoencoder

Encoder <-> Decoder

-> learns efficient compression / decompression
-> good for denoising
-> not concerned with labels
-> output dimension = input dimensions



MSE Loss is better!
Adam optimizer is better!

for data in train_loader:
    -> labels don't matter!

-> Compare images to reconstructions
    -> sometimes blurry -> Convolutional Autoencoder

Conv Autoencoder

-> encoder like normal CNN
-> decoder: Reverse Downsampling -> Unpooling
    -> Interpolation like Nearest Neighbor -> only copying -> learn Upsampling -> Transpose Convolutional Layers

(10)

Transpose Conv Layer

-> stride = downsample factor

-> use large filter than image

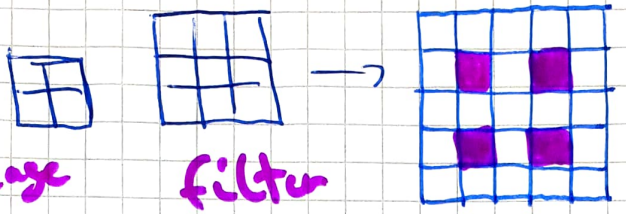


image 2×2 filter 3×3

Stride = 2
=> upsample 2x!

-> can throw artifacts, upsampling with Nearest Neighbor or Bilinear Interpolation

~~(or: Kernel size)~~

(ConvLayer)

-> upsample with resize to upsample 2x & normal conv instead of transpose conv layer

-> Denoising

Denoising
-> Deeper Networks are required
-> increase depth by 1 Conv layer in both Encoder & Decoder parts

Style Transfer

11

-> Merge Image with Style of another one

-> CNN extracts features, later layers: objects

-> Style: feature space for colors & shapes

=> see which features (color, shapes) are similar across filters

-> Style & content image

VGG19 is used

-> Conv stacks (4 Convs together)

Content Representation from content Image $\neq$ Content Representation from target image

↳ from Conv 4_2

Content Loss

$$L_{content} = \frac{1}{2} \sum (T_c - C_c)^2 \rightarrow \text{Minimize}$$

-> Should be close in representation even with different style!

-> Weights are not updated, only target img is updated (using backprop to minimize $L_{content}$)

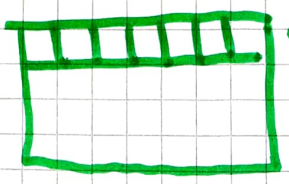
12

Same for target & style image

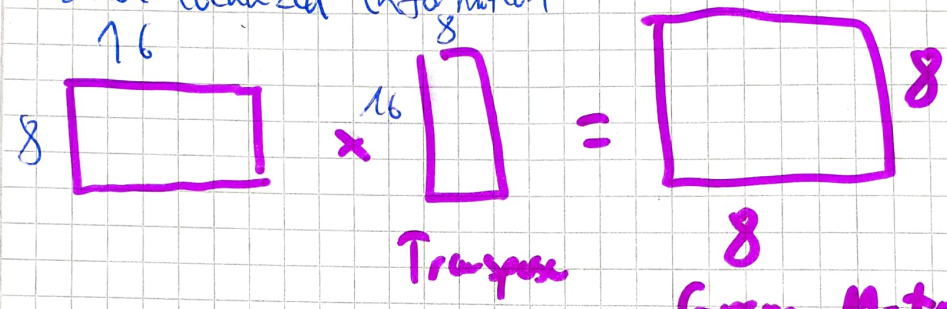
-> Multi scale Image in all 5 conv steps

Gram Matrix

-> flatten feature maps to go from 3D to 2D



-> non localized information



=> Multiply features in each map

-> rates style of target image

so

Style Loss

Mean Squared Error between style & target image Gram Matrices

$$L_{style} = a \sum_i w_i (T_{s,i} - S_{s,i})^2$$

T: Target Image, S: style Image

Total loss = $a L_{style} + L_{content} b$

a: content weight

b: style weight

→ requires_grad_(False)
→ freeze weights

Style Content Representations:

conv 1_1, conv2_1, conv3_1, conv 4_1, conv 4_2, conv 5_1

Style Loss:
→ iterate through style layers
→ same as content loss, but multiplied by weights

Content Loss:

= torch.mean((target_features['conv4_2'] - content_features['conv4_2'])**2)

$$Precision = \frac{TP}{TP + FP}$$

↑ Sensitivity

$$Recall = \frac{TP}{TP + FN}$$

↑ Specificity but with TN at the top

Input is 224 x 224:

222 → 220

220 → 216