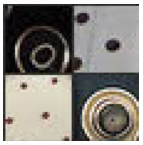
ADL4CV: Lecture Notes

Philipp Wulff / Summer semester 2022

ADL4CV - 1. Visualization

1

- Visualize the input space: (the receptive field)
  → Extract the input image patches that result in the highest activation for a feature map pixel/unit

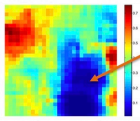
Layer 2 

  ⇒ In later layers the input image patches are more specialized (objects instead of lines)

Layer 5 

- Visualize importance (of features) (the input space):
  → Block different parts of the image and observe classification score change

Occlusion experiment

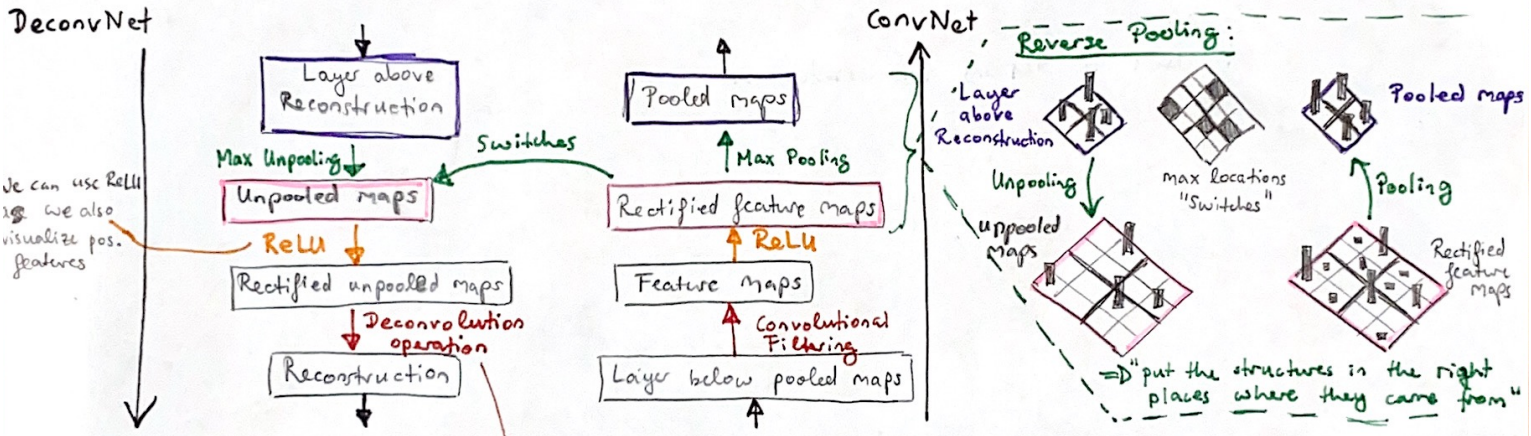


Most important pixels for classification

⇒ Create a map (each pixel is the classification prob. if an occlusion square is in this region)

- Visualize features: Map activations back to the image space / Use DeconvNet to visualize features at a certain layer (the layer)

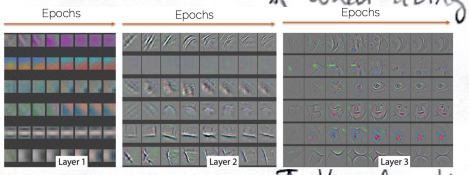
① Ex.: If filter 15 of the 3rd layer is highly activated
→ To visualize it: zero out other filters of layer 3 & pass through DeconvNet



Convolve with the transpose of the learned filter!
→ "tranposed" = vertically & horizontally flipped

⇒ print the reconstruction

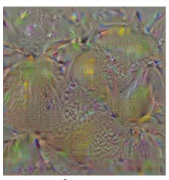
* When doing this for every filter of every layer → reconstructs
  → layer 1: low-level geom. features
  → deeper layers: reconstruct patterns / faces / cats / ...



After more epochs: Reconstructions become more detailed

↑ Visualization shows that large (e.g. 11x11) kernels produce inactive filters (→ empty/blurry reconstructions)

② With gradient ascent: generate a synthetic image that max. activates a filter.



E.g. find image that max. score of some class

1. Take trained CNN (data was zero mean)
2. Pass zero (empty) image through CNN
3. Max. score of class c with G.A. + Backprop
4. Update the image

$$\arg\max_I S_c(I) + \lambda \|I\|_2^2$$

L2-norm to avoid only a few large pixels

* Improve through regularization (Gaussian blur, clip small values in image or gradients)

⇒ Now, max. activations of feature maps instead.

③ Amplify the feature activations at some layer in the network (DeepDream)

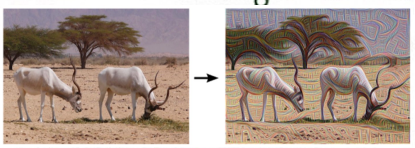
(-> on a trained classifier)

1. Forward pass of an image up to layer L
2. Set gradient of layer L = activations → large gradients for whatever was detected
   ⇒ show more "dogs"
3. Backprop (-> calculate the gradient of the activations of layer L w.r.t. the input image pixels)
4. Update image

⇒ Small changes to image according to a specific layer

- Low layers: Amplify basic geometries
- Deeper layers: Amplify objects

shallow layer:



Deep layer:

- Visualization with t-SNE:

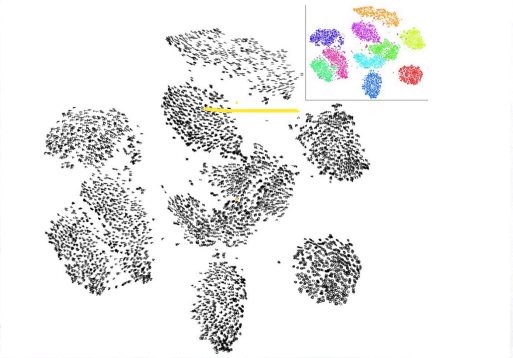
→ Visualize FC-layers (e.g. last FC-layer in AlexNet has dim. 4096)

↳ show clusters of images in feature space with t-SNE

* Maps high-dim embedding to 2D Map which preserves pairwise distances the points

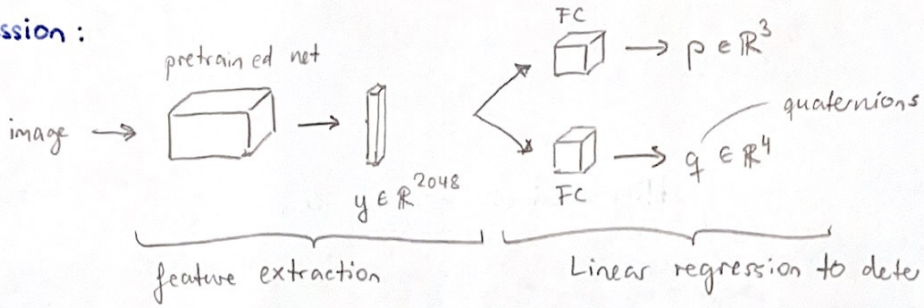
⇒ Used to debug the network

t-SNE Visualization: MNIST



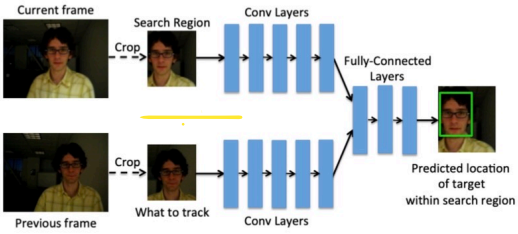
ADL4CV - 2. Siamese Neural Networks & Similarity Learning

- **Pose regression:**



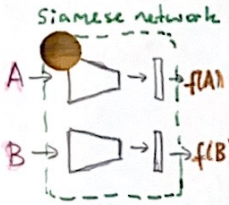
- **Bounding box regression:**

* Process current + previous frame with two feature extractors in parallel and achieve combined vector embedding → pass to FC-layers



- **Similarity Learning:** → Instead of discrete classification learn similarity function

→ Requires retraining whenever we add new samples



* Pass two images through Siamese networks (shared weights)

⇒ compare the encodings/embeddings

1) If A and B depict the same person: minimize $\mathcal{L}(A,B) = \|f(A) - f(B)\|^2$

2) If A and B depict different persons: minimize (Hinge-Loss) $\mathcal{L}(A,B) = \max(0, m^2 - \|f(A) - f(B)\|^2)$ (margin/threshold/constant)

If A and B are already m^2 apart, do not pay anymore attention.

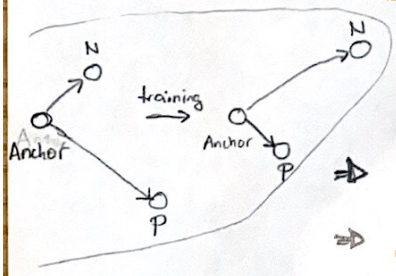
$$y^* = \begin{cases} 1, & \text{if A and B depict the same person} \\ 0, & \text{otherwise} \end{cases}$$

combines 1) and 2)

⇒ **Contrastive loss:** $\mathcal{L}(A,B) = y^* \|f(A) - f(B)\|^2 + (1 - y^*) \max(0, m^2 - \|f(A) - f(B)\|^2)$ (positive pair / negative pair)

Or use the **triplet loss:** We want $\|f(A) - f(P)\|^2 < \|f(A) - f(N)\|^2$

positive pair (depicts same person as A) / negative pair



⇒ $\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + m < 0$

⇒ $\mathcal{L}(A,P,N) = \max(0, \|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + m)$

Problem: - slow training when A and P are already similar

↳ Train only for a few epochs with the hard cases where $d(A,P) \approx d(A,N)$

↳ "Hard negative mining"

find from Nearest Neighbor search

* Random sampling does not work well -> O(n^3) possible triplets

⇒ triplet loss is more efficient & leads to better results for similarity learning

Since we learned a ranking -> at test time we only need to do nearest neighbour search

- Improving similarity learning:

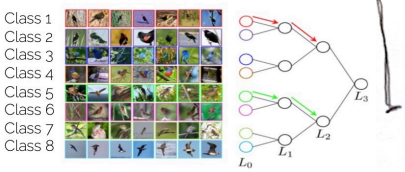
Loss: * Use triplet loss

Sampling: * Choose best triplets in training (diversity of classes + hard cases)

↳ In total $\mathcal{O}(n^3)$ combinations

→ Sampling method: **Hierarchical triplet loss**

1. Build the tree
   - Build tree where the leaves are the different image classes
   - Leaves are ordered s.t. its neighbor leaves are closest



$$d(p,q) = \frac{1}{n_p n_q} \sum_{\substack{i \in p \\ j \in q}} \|r_i - r_j\|^2$$

cardinality of classes p and q

feature embeddings of classes p and q

2. Sample diverse classes: Randomly select l' leaf nodes
3. For each of the l' chosen classes select the $m-1$ nearest neighbor classes
   -> learn hard cases
4. From each class randomly pick t images
   ⇒ $t \cdot m \cdot l'$ images per mini-batch
5. Construct all possible triplets from the mini-batch
6. Compute the triplet loss (the margin can be adaptable to the distances between classes on the hierarchical tree)

$$\mathcal{L}_M = \frac{1}{2Z_M} \sum_{T^z \in \mathcal{T}_M} [\|x_a^z - x_p^z\| - \|x_a^z - x_n^z\| + \alpha_z]_+$$

all the triplets

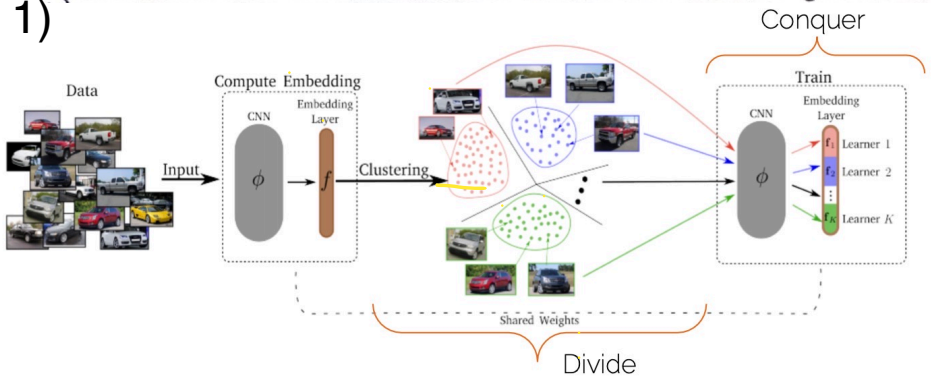
"Divide & Conquer"

Ensembles: *

DIVIDE
1) Cluster the embedding space into K clusters (e.g. with k-means)
2) Build k independent learners

3) Train each model on all mini-batches from its cluster

CONQUER
4) After convergence, finetune using all models at the same time
5) Since we updated the network, the embedding may have changed
   → repeat from 1)

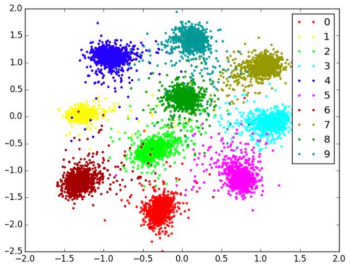


- Tips & Tricks:
  - Stronger backbone (feature extractor / dense net) improves results
  - Out-of-the-box: use ProxyNCA and SoftTripleLoss
  - Even naive ensembles boost performance (using the same architecture)

- Sampling strategy and choice of loss function are important

- Applications in vision:

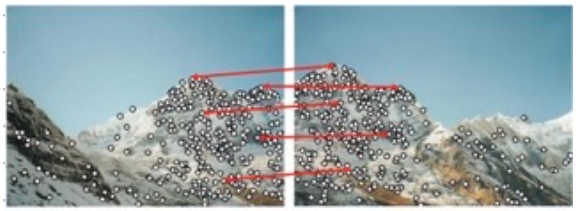
Classification: Siamese network on MNIST



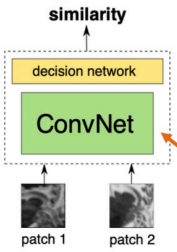
Establishing image correspondences

Classical method pipeline:

– Extract manually designed feature descriptors
  • Harris, SIFT, SURF: based on image gradients
  • suffer under illumination or viewpoint changes
  • Slow to extract dense features
– Match descriptors from the two images
  • Many descriptors are similar, one needs to filter out possible double matches and keep only reliable ones.



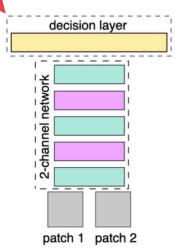
End-to-end learning for patch similarity



- Fast to allow dense extraction
- Invariant to a wide array of transformations (illumination, viewpoint)

Siamese network

- Classic Siamese architecture
  – Shared layers
    • Simulated feature extraction
  – One decision layer
    • Simulates the matching

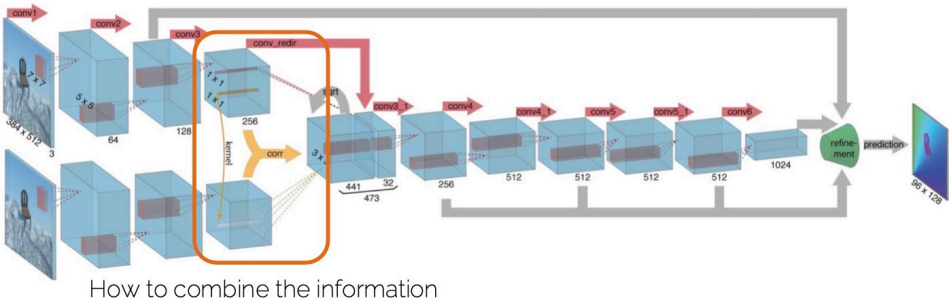


Optical Flow

- Input: 2 consecutive images (e.g. from a video)
- Output: per-pixel displacement from image A to B
- Result: “perceived” 2D motion (not the real motion)

-> e.g. FlowNet

- Architecture 1: Stack 2 images
- Architecture 2: Siamese network

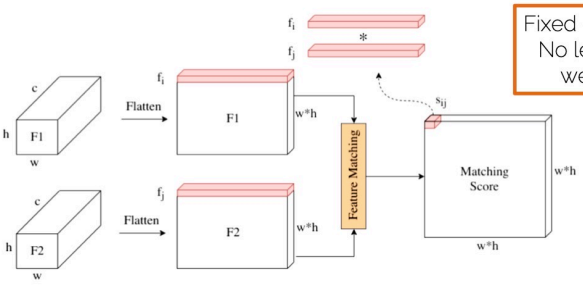


How to combine the information from both images?

Correlation Layer:

- Multiplies a feature vector with another feature vector

- The matching score represents how correlated these two feature vectors are

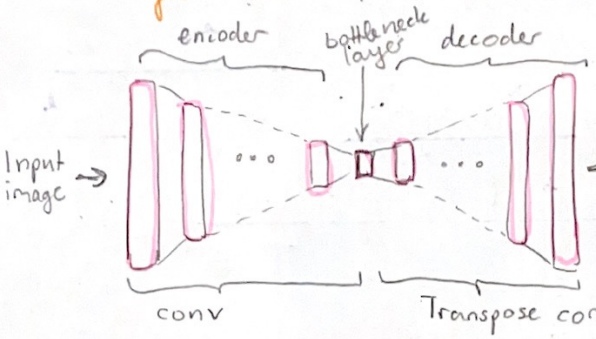


Fixed operation. No learnable weights!

ADL4CV 3 – Autoencoders & VAE

(3)

- **Autoencoders:** ↳ Unsupervised approach that learns a lower-dim. feature representation from unlabelled data



⇒ Use L1 or L2 loss as a reconstruction loss

Use-cases: * Use the latent representation for clustering

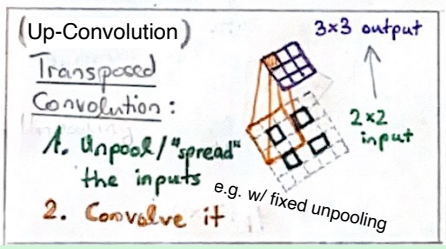
* For automatic pretraining → 1. Pretrain autoencoder on large unlabelled dataset
  2. Fine-tune the encoder with additional FC-layers on small labelled dataset

  encoder as feature extractor

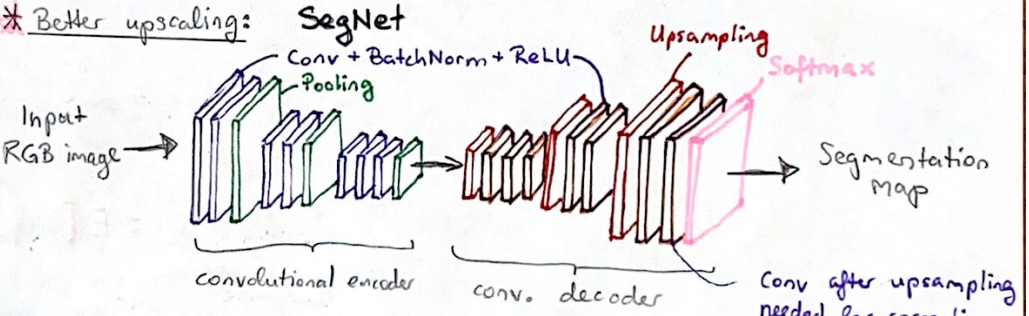
* For pixel-wise predictions:

→ **Semantic segmentation:**

* Previously used: fully-convolutional networks **(FCN)**
  ↳ 1. Encode image into latent representation
  2. Upscale it to the original resolution with a feature map for each class



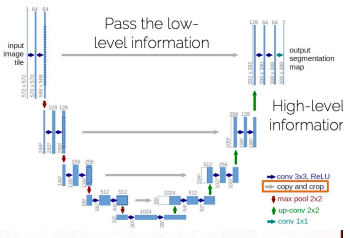
* Better upscaling: **SegNet**



Conv after upsampling needed for correction → refine upsampling

* Preserve low-level information with skip connections: **U-Net**

  * The encoder compresses high-level info in the channels of the bottleneck layer
  * Pass low-level info through skip-connections
    ⇒ crop feature maps from skip-connections such that dimensions align ⇒ concatenate
    ⇒ Results in sharp boundaries for segmentation



→ **Unsupervised monocular depth estimation**

1. Predict inverse depth map from left image
2. Use inverse warping to reconstruct the left image from right image + inv. depth map (general CV technique)
3. Min. reconstruction error

requires two cameras during training → only one at test time

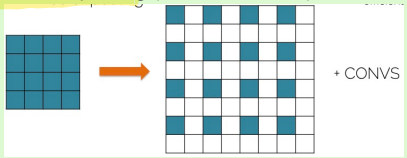
→ **Image Super Resolution**

* Add skip connection from input to last layer
  ⇒ the network only learns the residual needed for high-resolution.
  ⇒ No need to learn the image structure

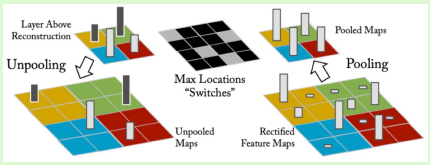
Upsampling Methods:

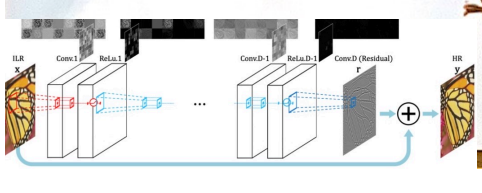
1) Interpolation: Nearest Neighbor/Lin./Bicubic
   -> few artefacts

2) Fixed Unpooling (+ Conv to refine) -> efficient

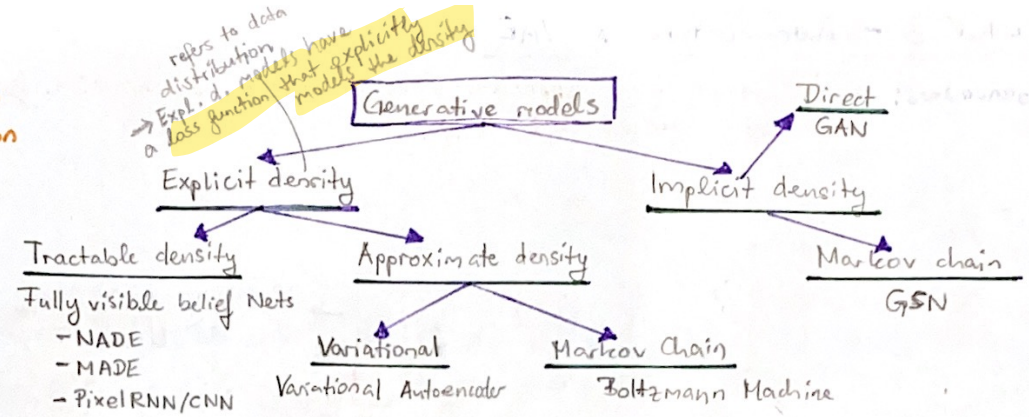


3) à la DeconvNet (save max. locations + Conv) (unlike in DeconvNet, Conv-weights are learned)
   -> keeps details of the structure

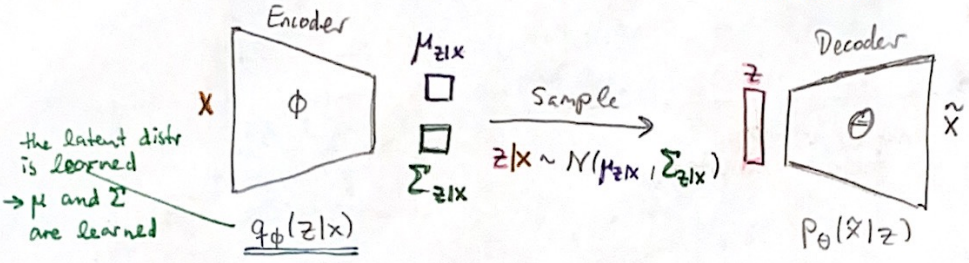




- Generative models:
  - ↳ create new samples from the same data distribution



→ Variational autoencoder: → Latent space is a distribution



At test time: Sample from latent space + pass through decoder

At training time: Optimize parameters ϕ and θ

↳ Loss function: $\log p_\theta(x_i) = \mathop{E}_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i)]$

"maximize the exp. log prob. of seeing the data point given the model parameters"

from Bayes' rule $p_\theta(z|x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)}$

$$= \mathop{E}_{z \sim q_\phi(z|x_i)} \left[\frac{p_\theta(x_i|z)\, p_\theta(z)}{p_\theta(z|x_i)} \right]$$

$$= \mathop{E}_{z \sim q_\phi(z|x_i)} \left[\frac{p_\theta(x_i|z) p_\theta(z)}{p_\theta(z|x_i)} \frac{q_\phi(z|x_i)}{q_\phi(z|x_i)} \right]$$

$$= \mathop{E}_{z} [\log p_\theta(x_i|z)] - \mathop{E}_{z} \left[\frac{q_\phi(z|x_i)}{p_\theta(z)} \right] + \mathop{E}_{z} \left[\frac{q_\theta(z|x_i)}{p_\theta(z|x_i)} \right]$$

definition of Kullback-Leibler divergence → measures difference between two distributions

likelihood — posterior — prior

$$= \mathop{E}_{z} [\log p_\theta(x_i|z)] - KL(q_\phi(z|x_i) \| p_\theta(z)) + KL(q_\theta(z|x_i) \| p_\theta(z|x_i))$$

- $\mathop{E}_{z} [\log p_\theta(x_i|z)]$: Reconstruction loss → Log-likelihood of reconstructing the data point given z
- $KL(q_\phi(z|x_i) \| p_\theta(z))$: Measures difference between latent distr. and Gaussian prior
- $KL(q_\theta(z|x_i) \| p_\theta(z|x_i))$: We do not know the shape of the distr. $p_\theta(x|z)$ → term is some value ≥ 0

$= \mathcal{L}(x_i|\phi,\theta)$ is a lower bound on the total loss

→ Optimize: $\phi^*, \theta^* = \arg\max_{\phi,\theta} \sum_{i=1}^{N} \mathcal{L}(x_i, \phi, \theta)$

Things to note: ✱ Latent space dimensions are interpretable
→ e.g. head pose, smile

* VAE Use-case: For generative models (—> sample from model)

- **Image Synthesis without GANs:**

(a) Input semantic layouts (b) Synthesized images

* Transform semantic segmentation image into real image

⇒ Can't use L2-loss between predicted image and ground truth image (it will penalize e.g. coloring white cars black, even though we only care about realistic colorization) → could penalize realistic results

⇒ Use perceptual loss: → measures the "content" of the image"

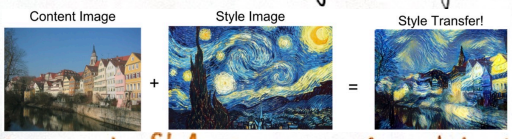
(a.k.a. content loss / feature reconstruction loss)

1. Take pretrained VGG network trained for image classification (→ ImageNet)
2. Pass synthesized & ground truth image through the network
3. Compare the feature maps / layers throughout the network

$$\ell_{feat}^{\Phi,j}(\hat{y}, y) = \frac{1}{C_j H_j W_j} \| \Phi_j(\hat{y}) - \Phi_j(y) \|_2^2$$

($\hat{y}$: feature maps of gen. image at layer j generated; $C_j H_j W_j$: feature map size (channels, height, width); $\Phi_j(y)$: feature maps of the g.t. image at layer j)

⇒ Intuition: the image classification network will produce "similar" features in deeper levels for images with the same object (e.g. car) independent of fine details (e.g. color)



- **Style transfer:** Content image + Style image → style transf. image

⇒ Use style loss: → measures the style but not the content

1. – " – } as in content loss
2. – " –

3. Compare the Gram matrices of feature maps throughout the network

$$\ell_{style}^{\Phi,j}(\hat{y}, y) = \| G_j^{\Phi}(\hat{y}) - G_j^{\Phi}(y) \|_F^2 \quad \text{— Frobenius Norm}$$

$$\text{with } G_j^{\Phi}(x)_{c,c'} = \frac{1}{C_j H_j W_j} \sum_{h=1}^{H_j} \sum_{w=1}^{W_j} \Phi_j(x)_{h,w,c} \cdot \Phi_j(x)_{h,w,c'}$$

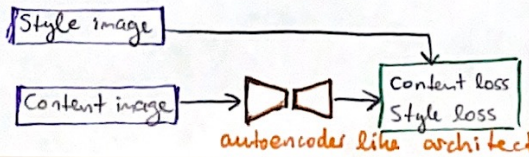
(c, c': comparing channels c and c')

⇒ Intuition: Captures information about which features tend to activate together

* Option to combine loss functions: Content loss + Style loss
w.r.t. content image w.r.t. style image

⇒ Iteratively update white noise image to take on content + style (slow)

* Fast Neural Style Transfer:
↳ In training: Use same style image & different content images



⇒ At test time: Only one forward pass needed.

ADL4CV 4 - Graph Neural Networks (GNN)

* Different data domain:

- e.g. convolutions require a certain structure (→ domain is regular) (order of pixels matters)

=> GNNs work on new domain: - Irregular
- Permutation invariant (order of points does not matter)
- Transformation invariant (e.g. the prediction is the same for a rotated point cloud)

e.g. - point clouds
- citation networks (node = paper, conn = citation)
- social networks

* Key challenges:

- Variable input size (# of nodes and edges)
- Need invariance to node permutations (order in which nodes are processed)

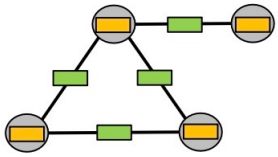
* General idea: Encode contextual graph info in node embeddings by iteratively combining neighboring nodes' features

"Each update step is understood as a layer in common NNs"

Graph: $G = (V, E)$ (vertices/nodes, edges)

Initial node embeddings: $h_i^{(0)}$ with $i \in V$

Initial edge embeddings: $h_{i,j}^{(0)}$ with $(i,j) \in E$



Initial Graph

At iteration step ℓ: (Method only w/ node embeddings)

collect info from all neigh.

① Message passing: $m_v^{(\ell+1)} = \sum_{u \in N_v} M^{(\ell)}(h_u^{(\ell)}, h_v^{(\ell)}, h_{(u,v)}^{(\ell)})$

(own embedding, neighbor embedding, edge embedding)

aggregation over all neighbors (order invariant)

$M^{(\ell)}$: Learnable function w/ shared weights across graph

② Embedding update: $h_v^{(\ell+1)} = U^{(\ell)}(h_v^{(\ell)}, m_v^{(\ell+1)})$

"only learns nodes"

$U^{(\ell)}$: Learnable function with shared weights across graph

There are different types of GNNs depending on the formulation of M(…) and U(…).

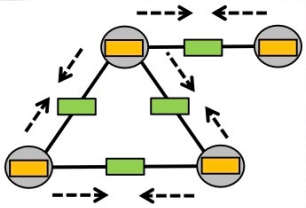
example: Graph Conv. NN

$$m_v^{(\ell+1)} = \sum_{u \in N_v \cup \{v\}} \frac{h_u^{(\ell)}}{\sqrt{|N_v| \cdot |N_u|}}$$

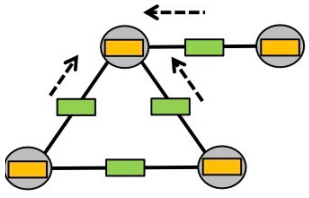
(also containing embedding of node v itself; also loops over itself; Normalization)

$$h_v^{(\ell+1)} = \sigma(W^{(\ell+1)} m_v^{(\ell+1)})$$

(σ: non-lin.; $W^{(\ell+1)}$: learnable matrix with size (# channels out, # channels in))



'Node to edge' Update



'Edge to Node' Update

Method with Node + Edge updates:

① Node-to-edge updates:

$$h_{(i,j)}^{(\ell+1)} = N_e(h_i^{(\ell)}, h_{(i,j)}^{(\ell)}, h_j^{(\ell)})$$

(N_e: learnable func. with weights shared across graph)

② Edge-to-Node updates:

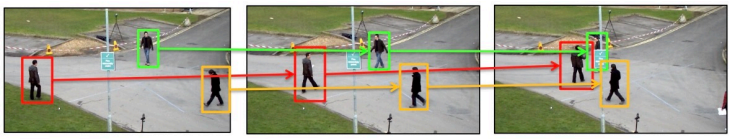
$$m_i^{(\ell+1)} = \Phi(\{h_{(i,j)}^{(\ell+1)}\}_{j \in N_i})$$

(Φ: order invariant operation (sum/...); N_i: neighbors of node i; edge embeddings)

$$h_i^{(\ell+1)} = N_v(m_i^{(\ell+1)}, h_i^{(\ell)})$$

=> Obtain edge and node embedding that contain contextual information about the graph topology and neighbor's features

- Message passing networks for CV:



* Multi-object tracking: 1. Object detection
  ↳ every object is node and each edge can be a trajectory
  2. Data Association
  ↳ connect nodes -> Use the GNN to find the best connections

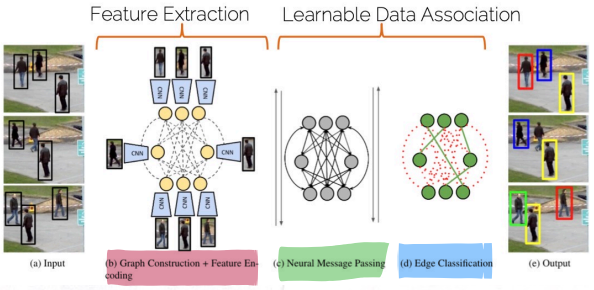
→ In Step 2.:

initial embeddings

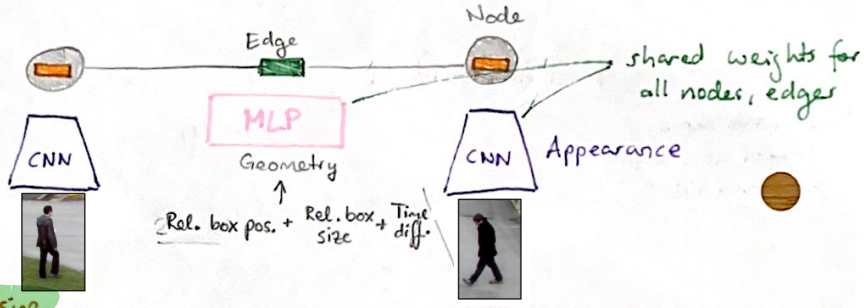
(b) Encode appearance and scene geometry cues into node & edge embeddings (e.g. distance)

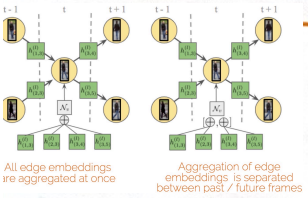
(c) Propagate cues

(d) Classify edge embeddings → predict solutions of the tracking problem



→ Feature encoding





→ Time-aware Message Passing
→ Separate the aggregation of edge embeddings between past / future frames

→ Classifying edges "Classify into active and inactive edges"
→ use MLP with edge embeddings as input -> predict whether an edge is active/inactive

Loss function:

$$\mathcal{L} = \frac{-1}{|E|} \sum_{l=l_0}^{L} \sum_{(i,j) \in E} w \cdot y_{(i,j)} \log\left(\hat{y}^{(l)}_{(i,j)}\right) + (1 - y_{(i,j)}) \log\left(1 - \hat{y}^{(l)}_{(i,j)}\right)$$

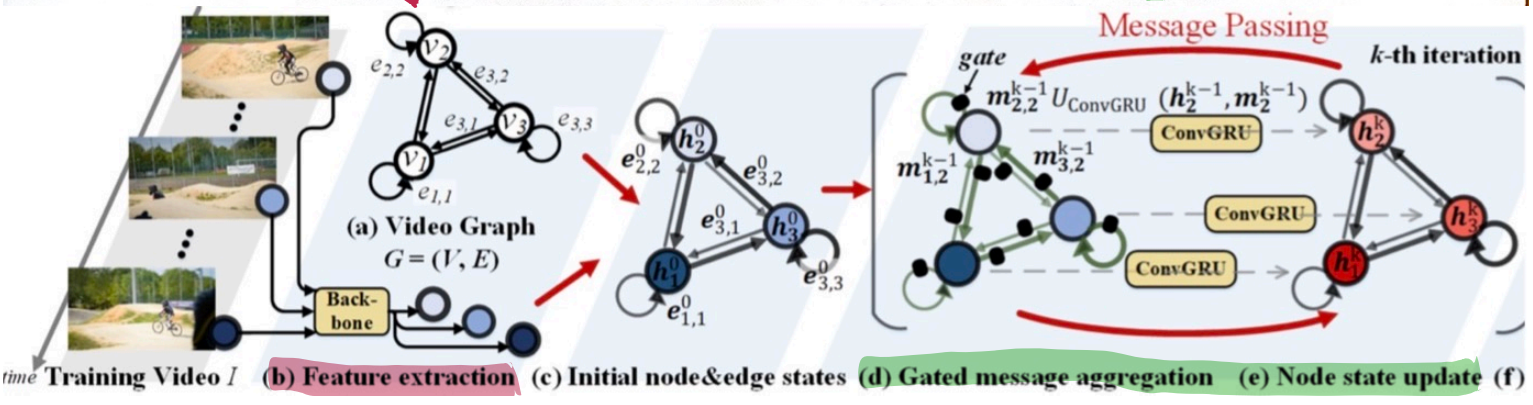
- $l = l_0$ to L: sum over last steps
- w: weight to balance active/inactive edges
- $\hat{y}^{(l)}_{(i,j)}$: edge prediction w/ sigmoid at iteration l
- BCE-Loss

* Video object segmentation: -> Goal: Generate accurate and temporally consistent pixel masks for objects in a video sequence.

→ If only one object to segment ⇒ Each node is a frame

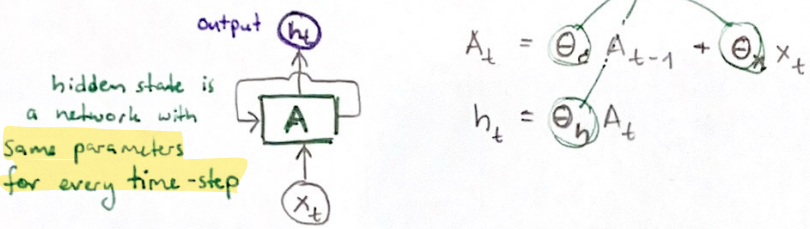
* Initial embeddings: extract features with DeepLabV3

* Message passing with conv. recurrent network to preserve spatial information (embeddings are feature maps) ⇒ keep pixel outputs

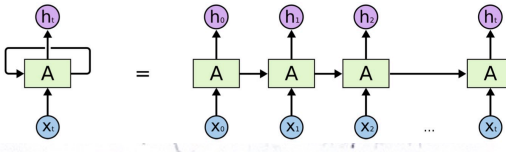


~~Attention~~:

- RNN : → Notion of "time"/"sequence"

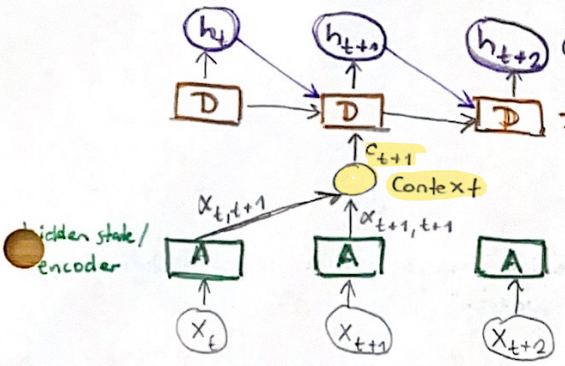


Unrolling a RNN:



⇒ Problem with long-term dependencies (which input is important for output h_t)

- Attention : → Aims to solve the long-term dependency problem



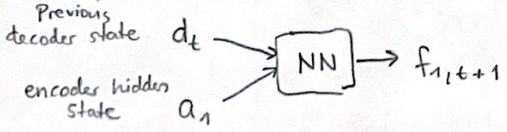
Input to decoder is:
- previous decoder hidden state
- previous output
- attention

⇒ $\alpha_{1,t+1}$ = how important input 1 is for output $t+1$

with $c_{t+1} = \sum_{k=1}^{t+1} \alpha_{k,t+1} \cdot a_k$ (weight · hidden state)

* Soft Attention = all attention masks / weights sum up to 1.

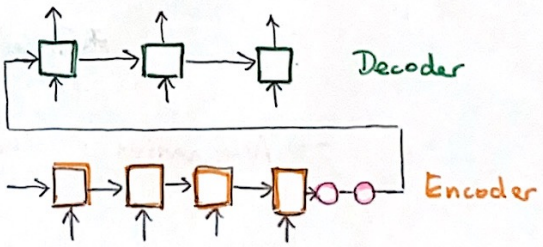
* To compute the attention mask : (Requires training a small NN)



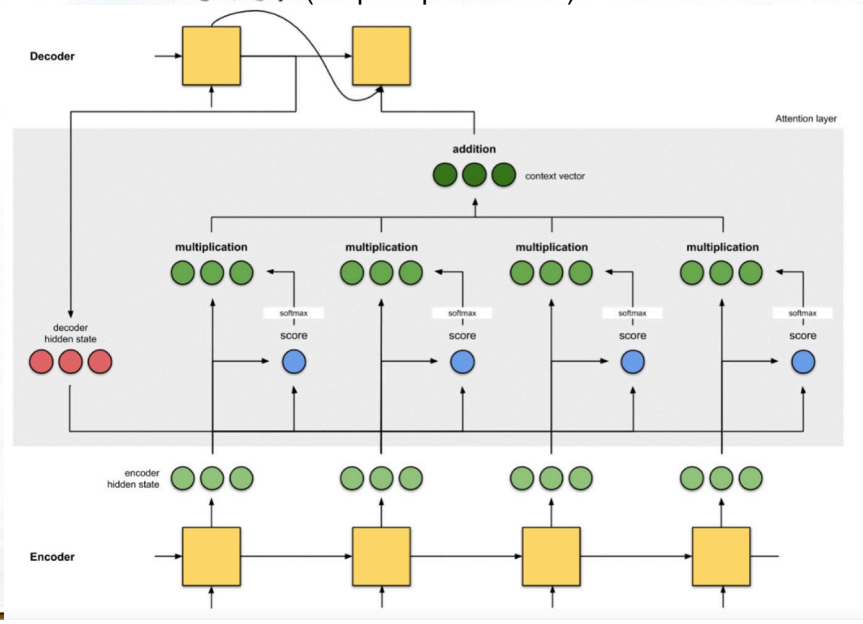
Normalize ⇒ $\alpha_{1,t+1} = \frac{\exp(f_{1,t+1})}{\sum_{k=1}^{t+1} \exp(f_{k,t+1})}$

- Seq2Seq : → Network for sentence translation

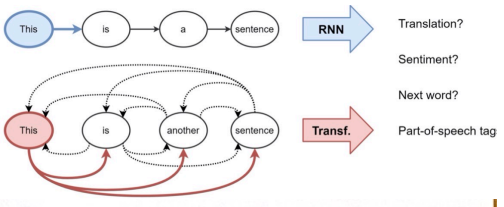
1. Read the whole sentence in language 1 and embed it
2. Translate it to language 2 using embedding + each word



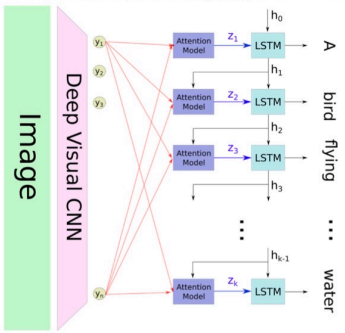
⇒ Also use Attention (Seq2Seq + Attention)



- Transformers: → No RNN, just attention.
  - * Current SotA in NLP
  - * Are basically a Graph Attention Network (GAT)
    - → GNN where the aggregation step is done via attention (weighted sum)



- Attention for CV:
  - * Image captioning
    1. Input: image
    2. Encode the image (feature extractor/CNN, e.g. AlexNet)
    3. Decoder: Attention-based RNN
    4. Output: Sentence describing the image



$\mathbf{y_i}$: Output of encoder are the image features which still retain spatial information (no FC layer!)

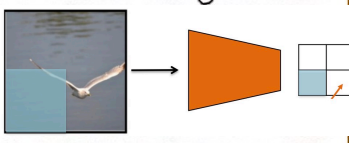
$\mathbf{Z_i}$: Output of attention model

$\mathbf{h_i}$: Hidden state of LSTM

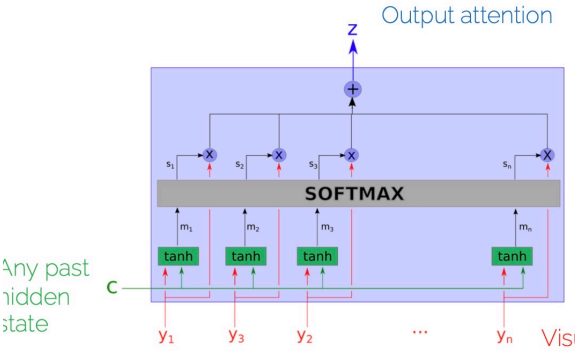
(Soft-)
→ Attention-mechanism: "focuses on a specific image patch when outputting a word."

Why do we need Attention?

Inputs to the Attention model: Feature descriptor for an image patch (→ feature map)

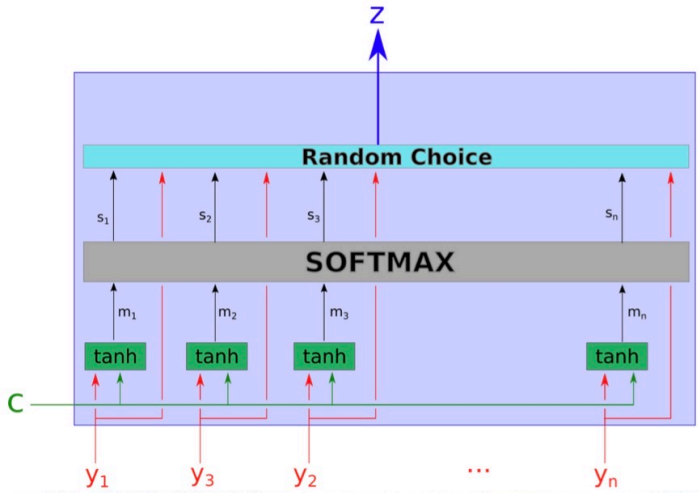


→ tanh to create bounded outputs
→ softmax for attention values between 0 and 1
→ Multiply by image features to rank them by importance



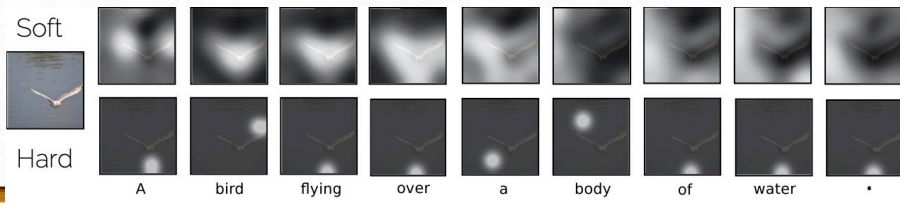
* Process is deterministic and can be backproped

→ Alternative: Hard-Attention: stochastic process; gradient via Monte Carlo sampling + estimation



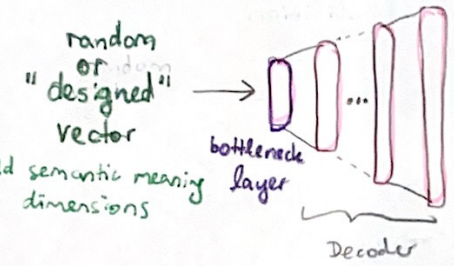
Chose features by sampling with probabilities s_i

- Soft vs hard attention



ADL4CV 5 - Generative Neural Networks

- Decoder as Generative model : → train decoder directly



→ add semantic meaning to dimensions

(L2 → blurry images as "mean" is best)

- Generative Adversarial Networks (GANs) : → Learn the loss function D

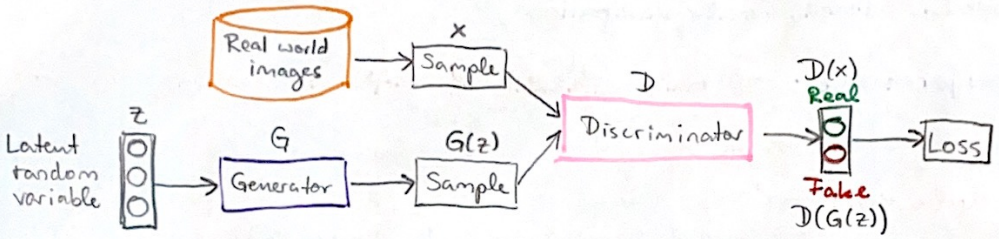
* is an implicit density model = the loss function does not explicitly match the data distribution

On real data:

$D(x)$ tries to be near 1

On fake data:

D tries to make $D(G(z))$ near 0, G tries to make $D(G(z))$ near 1



→ Alternating gradient updates:

"First debug step: Only do 1). If $J^{(D)}$ can't go to 0 something is wrong."

1) Fix G and perform gradient descend on the discriminator loss :

from BCE loss

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}} \log D(x) - \frac{1}{2}\mathbb{E}_{z} \log(1 - D(G(z)))$$

→ the equilibrium is a saddle point of the discriminator loss

2) Fix D and perform the gradient descend on the generator loss :

G min. the prob. that D is correct

$$J^{(G)} = -J^{(D)}$$

"Minimax game loss"

or G max. the log prob. of D being mistaken

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_{z} \log D(G(z))$$

used in practice

→ G can still learn even if D rejects all the gen. samples

"Heuristic method loss"

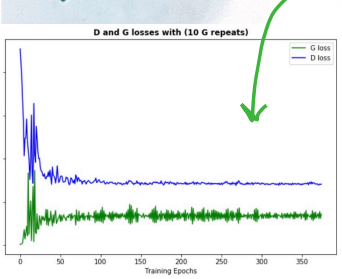
* E.g. perform updates until on loss goes down, then alternate (or other schedules)
* Joint updates also possible

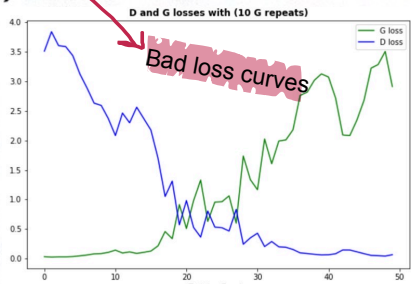
→ Loss functions :

"D- and G- loss must be balanced → D provides supervison for G"

* If the D loss gets to 0, there is no recovery
* Good loss curves = plateau at constant value (≠ 0)
* The larger the image, the more difficult the generator architecture becomes
* Cannot determine if training is done from loss
  ↳ Look at generated samples
* Training schedules :
  → Adaptive schedules e.g. { while loss_D > t_D : train D ; while loss_G > t_G : train G

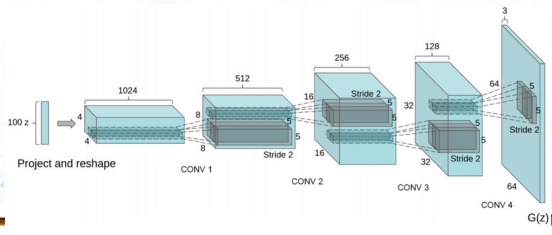
If D rejects all samples from G, the G-loss is always high and D-loss is always low -> G cannot learn





deep convolutional GAN

- DCGAN : → 2015 Paper ; Is a good baseline architecture



- **Mode Collapse:** → is the Problem when the generated images fool D, but have little variety.

$$\max_D \min_G V(G,D) \neq \min_G \max_D V(G,D) = \min_G \max_D \underbrace{\mathbb{E}_{x\sim p_{data}} \log D(x) + \mathbb{E}_z \log(1-D(G(z)))}_{\text{value function definition}}$$

G wants to min. the value func. / D wants to max. it

loop ordering
- * if D is in the inner loop → converges to correct distr.
- * if G is in the inner loop → D is fixed while updating G → "learn to fool the same D" → easy to converge to one sample

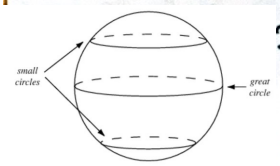
Additional observations: (modes = faces, cars, etc. in the data)
- * More modes → smaller recovery rate
- * Larger latent space → more mode collapse

⇒ For these reasons, GANs are usually trained on specific domains (smaller distributions)

- **Evaluation of GAN performance:** → main difficulty of GANs is knowing how good they are
  - * GANs have problems with generating global structures (e.g. counting)
  - * **Human evaluation:** → check training curves
    → predict same latent codes every n steps → variety / become more realistic
  - * **Inception score (IS):** 1) Train a classifier on the real images & classify generated images
    2) Saliency: Check if images are classified with high confidence
    3) Diversity: Check if we obtain images from all classes
  - * **Look at discriminator:** → Use D-layers for feature extractor in a classifier
    → fine-tune the last layer
    → if high class accuracy → we have good D and G

… since Convs are "local" operators

- **GAN Hacks:**
  1) Normalize inputs between -1 and 1 + tanh as last generator layer → so that all inputs to the discriminator are in the same range
  2) Don't sample z from uniform distr. / Sample z from Gaussian or from surface of hypersphere → Do interpolations via a great circle rather than linearly



  3) Try BatchNorm; Construct mini-batches with all real or all fake images → stabilizes training
  4) Adam for generator, SGD for discriminator
  5) **One-sided label smoothing:** $J^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}} \lambda \log D(x) - \frac{1}{2}\mathbb{E}_z \log(1-D(G(z)))$ (λ = some value, λ < 1)
     → If image is real (y=1) set $y = \lambda$ to some value < 1
     → Reduces confidence of D
  6) **Historical Generator batches:** Construct batch from current + past generated images → pass to D; Stabilizes D
  7) **Avoid sparse gradients:** → Use LeakyReLU instead of ReLU
     → Downsampling: Avg. pool or conv + stride
     → Upsampling: deconv + stride or PixelShuffle
  8) **Exp. averaging of weights:** keep second 'vector' of weights that are averaged -> almost no cost, average of weights from last n iters

Problem: D is noisy due to SGD

- Other objective functions: → Don't change it too much; mostly the heuristic loss is fine

The loss function alone will not make it suddenly work!

→ EBGAN: Energy-based Generative Adversarial Networks

* discriminator is an autoencoder trained on real images

$$D(x,z) = D(x) + [m - D(G(z))]^+ \quad (D(x) \to 0,\ D(G(z)) \to m)$$

$$\mathcal{L}_G(z) = D(G(z)) \quad (D(G(z)) \to 0)$$

→ want reconstruction cost $D(x) = \|Dec(Enc(x)) - x\|$ to be low

⇒ if there is a good reconstruction, it is a real image

where $[u]^+ = max(0, u)$

- We want to penalize the discriminator if the reconstruction error for generated images drops below a value m.

→ BEGAN: Boundary equilibrium GAN

* Similar to EBGAN, but instead of a reconstruction loss measure the difference in the data distr. of real and generated images

↳ Wasserstein distance / Earth-mover distance

→ WGAN: Wasserstein GAN

* BEGAN with Wasserstein distance (comp. dist. between two distributions)

→ Problem: Wasserstein-dist. is expensive to compute

↳ Get upper bound from dual: $W(P_r, P_\theta) = \sup_{\|f\|_L \le 1} \mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_\theta}[f(x)]$

f is 1-Lipschitz function

f is difficult to define → approximate f by learning a NN

1-Lipschitz constraint: $|f(x_1) - f(x_2)| \le |x_1 - x_2|$

⇒ Define Update with weight clipping to train a Lipschitz-function:

$w \leftarrow w + \alpha \cdot RMSProp(w, g_w)$

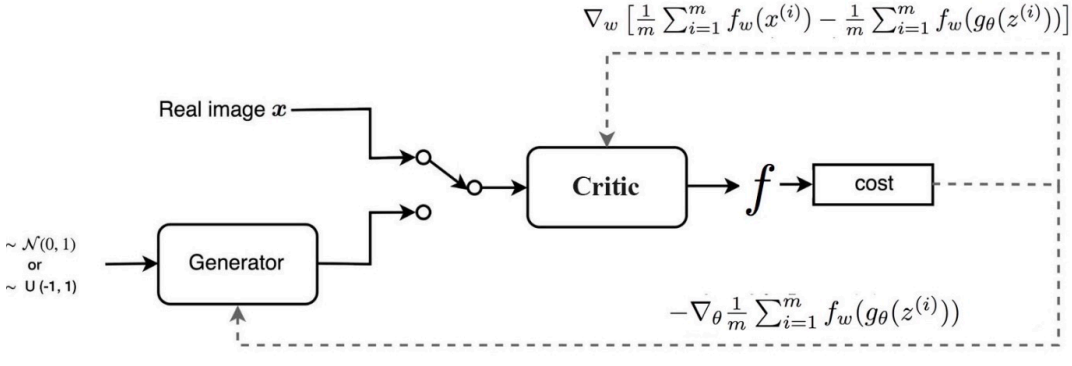
$w \leftarrow clip(w, -c, c)$

→ f is the critic func

⇒ Main advantage of WGAN: Discriminator/Critic loss converges to zero instead of plateauing

+ mitigates mode collapse
+ generator still learns when critic performs well
+ actual convergence

- Enforcing Lipschitz constraint is difficult
- Weight clipping is "terrible"
  -> too high: takes long time to reach limit; slow training
  -> too small: vanishing gradients when layers are big



ADL4CV 6 - GAN Architectures & Conditional GANs

Multiscale GANs → deal w/ the problem where generating the global image structure is difficult

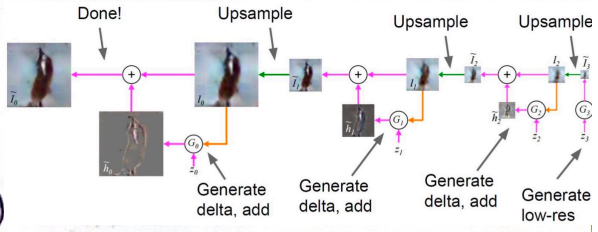
↳ from 2015

Idea: Multiple generators at different image scales w/ upscaling in between

↳ each higher-scale generator adds details to the current image

(Generators at low-res learns the global structure)

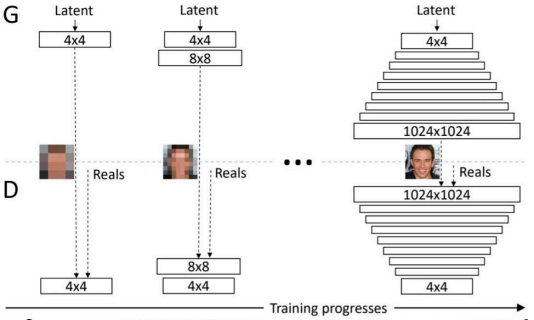
→ We need discriminators at every level/resolution



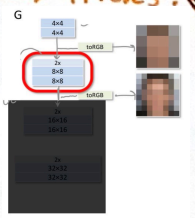
Progressive Growing GANs → from 2017; extends Multiscale GANs

1. Start by generating low-res images + train
2. Append upsampling block to G and downsampling to D
   ↳ higher resolution; initialize weights in lower layers w/ previous weights
3. fixed schedule when to append the next layer

train processes + 1



Additional Tricks:



- 1-layer feature maps (get RGB-image from series of 1x1-convs)
  ↳ makes appending upsampling blocks easier → "to-RGB" operator
- When adding new blocks: lin. crossfade between "old" & "new" Image reconstr. loss
  ↳ gradually move weight to larger layer ⇒ stabilizes the loss
- Analogeous "from-RGB" operator for discriminator

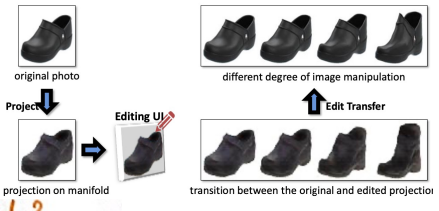
Conditional GANs → for Modeling (adding semantic meaning to the output / "what to generate")

(cGANs)

Naive approach: Use condition img as input to G

Problem: D only judges on real/fake but not if output img ≈ cond. img

↳ iGAN: 1) project real image onto learned latent manifold
2) Edit projection for example (→ edit latent code)
3) "Generate" image w/ modified latent code



→ How to project a real image onto the manifold?

with optimization $z^* = \arg\min_z \mathcal{L}(G(z), x^R)$

z: latent vector; $\mathcal{L}$: reconstr. loss; G: trained generator; x^R: real image

→ instead of solving the optimization problem, train an inverting network $z^* = P(x^R)$

$$\theta_P^* = \arg\min_{\theta_P} \sum_{x_n^R} \mathcal{L}\left(G(P(x^R, \theta_P)), x^R\right)$$

$G(P(x^R, \theta_P))$: like an auto-encoder w/ a fixed decoder G

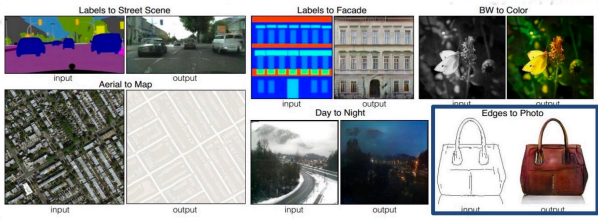
Hybrid method: first use inverting network, then fine-tune by solving the optimization problem

↳ In practice they add a manifold smoothness loss.

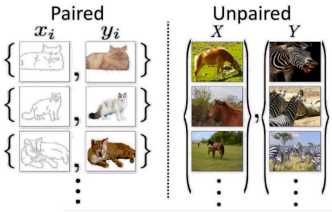
Edit transfer in the OG paper: reproject edits in the generated image onto the real image -> fewer imperfections

Objective: $z^* = \arg\min_z \left\{ \sum_g \mathcal{L}_g(G(z), v_g) + \lambda_s \|z - z_0\|_2^2 \right\}$

user guidance; stay close to some real image



Paired vs. unpaired setting:



Paired: x_i and y_i can be inferred from each other or are available as a pair

unpaired: X and Y are not available as a pair
↳ inference of Y. is the task

Make the discriminator aware of the user guidance (s.t. it has an influence on the generated image)

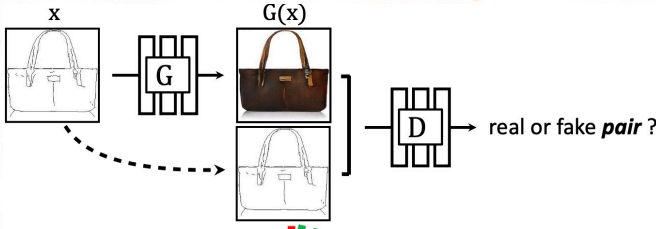
↳ Need paired data (often called self-supervised) (e.g. edge detection)

↳ true conditional GAN

↳ match joint distribution

$P(G(x)_1 y) \sim P(x, y)$

→ E.g. Pix2Pix (2017)



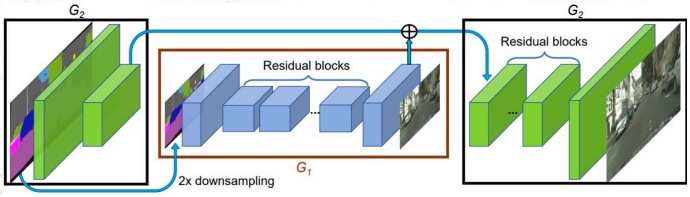
$$\min_G \max_D \mathbb{E}_{x,y}[\log D(x, G(x)) + \log(1 - D(x, y))]$$

fake pair — real pair

↳ Tricks in Pix2Pix:
- adds L1-loss (GT & output image) loss to GAN-loss, since the problem is underconstrained
  ↳ $L = L_{GAN} + \lambda L_1$
- use dropout to add noise → probabilistic model

Pix2PixHD (2018) —> cGANs + Multi-scale

"Generator within a generator"
→ it is more efficient to use the weights in this hierarchical architecture (You can only fit so many weights in a GPU)



⇒ Conditioning at multiple scales (in the image at 2 resolutions)

How to formulate a conditional GAN in the unpaired setting?

↳ no input-output pairs

Problem: We have not input-output pairs anymore
-> how to make G care about the input?

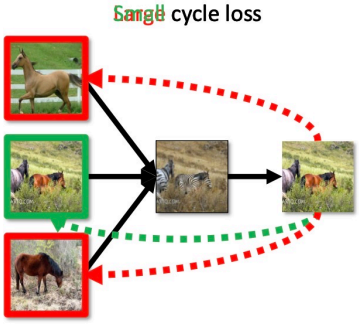
This will probably lead to mode collapse

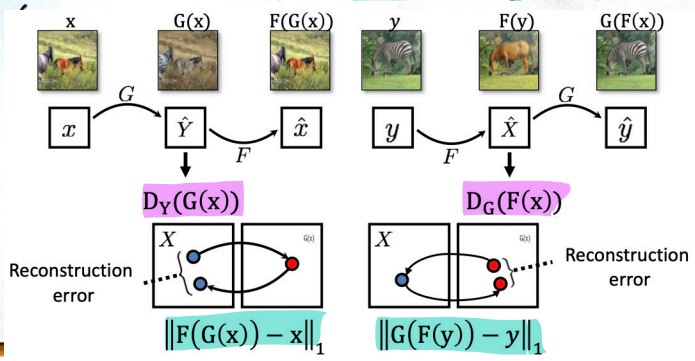
- We want a bijective mapping (each input points at exactly one output)
- Method: Enforce Cycle-Consistency (mapping from X to Y results in unique Y and the other direction gives unique X)

Cycle Consistency Loss:

↳ Use another generator F to map Y to X

→ Discriminator loss + Reconstr. loss (In each direction -> 4 losses in total)





ADL4CV 7 - Videos & Autoregression

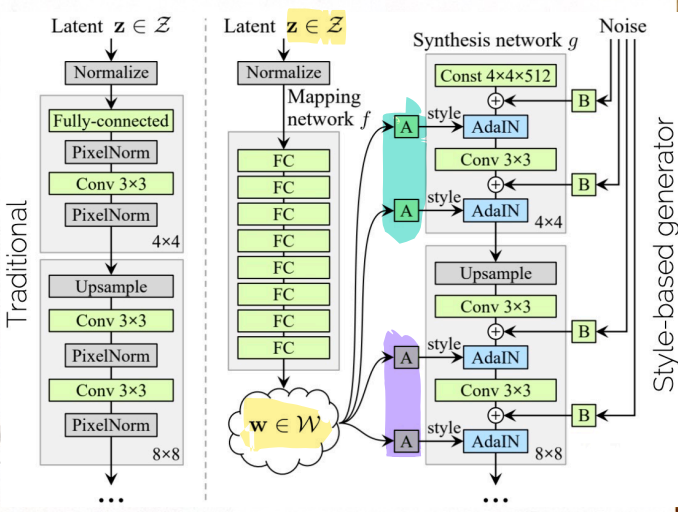
StyleGAN (2019)

↳ Difference to traditional GAN:

use style image as input to mapping net →

* Mapping network maps latent code z to an interpretable style vector w
* Feed in noise & style vectors every few layers (they use AdaIN) → added to feature map
* Style vectors added at low-res are coarse styles (thereafter middle + fine styles)

—> multi-resolution control (e.g. change face details independent of main features)



Autoregressive Models

* GANs learn implicit data distribution (→ outputs are samples & distribution is in the model)
* Autoregressive models learn an explicit distribution (⇔ outputs are probabilities)

governed by a prior imposed by the model structure

↳ PixelRNN (2016)

* Interprets image pixels as a sequence
  → product of conditional distributions
  → Next pixel determined by all previous pixels

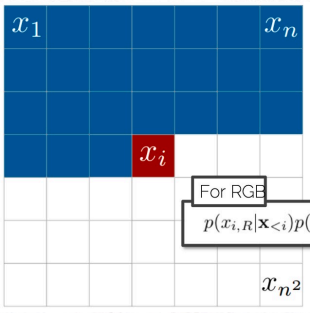
$$\Rightarrow p(\vec{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, \dots, x_{i-1})$$

prob. of all pixels (in practice = softmax w/ 256 options)

conditioning on previous pixels

→ over complete image

e.g. prob that a pixel is green

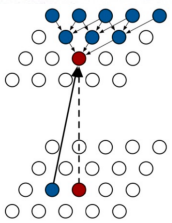


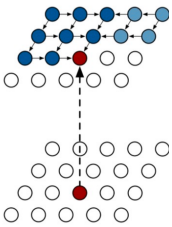
$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, \dots, x_{i-1})$$

For RGB

$$p(x_{i,R}|\mathbf{x}_{<i})p(x_{i,G}|\mathbf{x}_{<i}, x_{i,R})p(x_{i,B}|\mathbf{x}_{<i}, x_{i,R}, x_{i,G})$$

* To reduce the sequence length: Row LSTM model architecture
  ↳ Each pixel depends on "pyramid" of pixels above
  → Image can be processed row by row ⇒ on GPU
  → However, incomplete context for each pixel
    ↳ Solution: Diagonal BiLSTM model architecture
    ⇒ diagonal pixels can be processed in parallel





→ PixelCNN (2016) → to lower computational cost

* Masked convolutions to avoid seeing future context
  → mimics sequence behavior

Mask A

* Specialty: Replace ReLUs with Gated Blocks

$$y = \tanh(W_{k,f} * x) \odot \sigma(W_{k,g} * x)$$

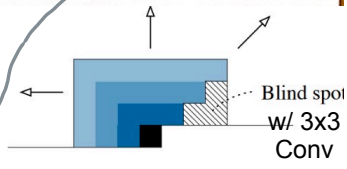
sigmoid

weights in kth layer

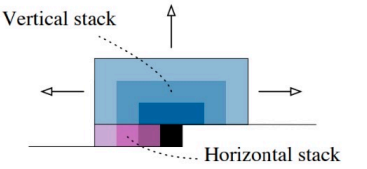
element-wise product

convolution

↳ mimics the LSTM



* Problem: Unseen context for pixel (Blind spot)
  ↳ Split convolutions in two stacks



Autoregressive models vs. GANs

⊕ more stable since probabilistic densities are modelled explicitly
→ easier hyperparam-tuning
⊕ applicable to discrete & continuous data

⊕ Demonstrations could produce higher quality images
⊕ faster to train (But also harder to train)

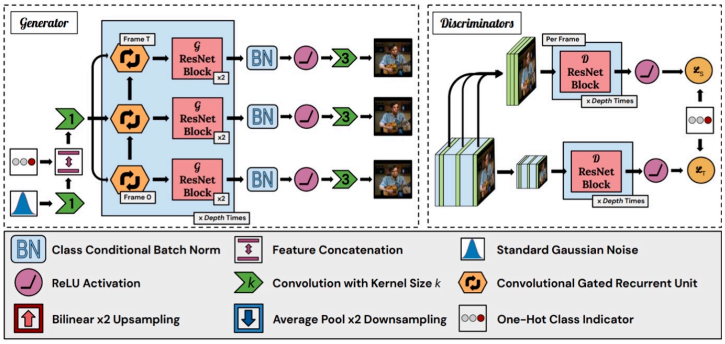
Generative Models on Videos

Two options:
- Single latent code for all frames (⊖ future frames deterministic given past)
- Latent code per frame (⊖ Need conditioning of future frames on past frames)

⇒ General Issues: Temporal Coherency * Collapse to mean image
(Drift over time, since the most recent frames are also generated and not GT anymore)

→ DVD-GAN (2019) (not discussed in detail)

* Single random vector seeds a video
* Per-frame generators
* Discriminators judge if the sequence is reasonable



Challenge: cGANs provide high-quality frames, but they are temporally inconsistent.

→ Conditional GANs work much better on videos
↳ the labels are guidance for temporal consistency if they are so themselves

→ Video2Video Synthesis (2018)
↳ use sequential generator $p(\vec{x}^T \mid \vec{s}^T) = \prod_{t=1}^{T} p(x_t \mid \vec{x}_{t-L}^{t-1}, \vec{s}_{t-L}^{t})$ (past L generated frames; past L source frames)

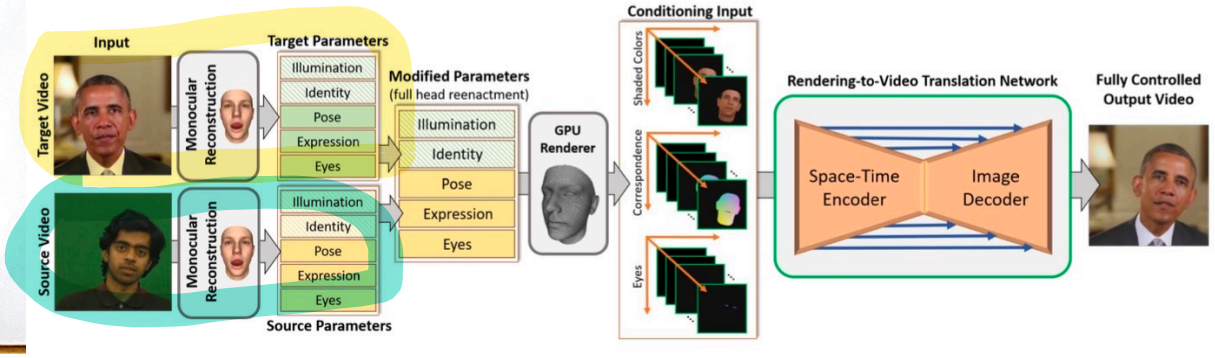
in practice: L = 2

Two Discriminators:
* Conditional img D (is img real?) + conditional vid D (temp. consistency) based on optical flow

→ Deep Video Portraits (2018)
* Train a cond. GAN on a target video w/ face tracking info as guidance (Rerendering)
(tracking quality is very important)
* At test time use guidance from source video

—> if the source video is temporally consistent the output video will be temporal consistent

⇒ Synthetic data for tracking is a great anchor —> Needs to stay within the training set w.r.t. motions

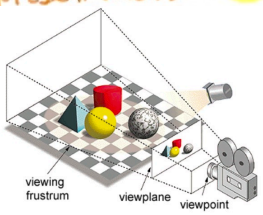


"Everybody Dance Now" has the same concept, but conditions on tracked body poses and generates dance videos

ADL4CV 8: Neural Rendering

11

- What is rendering? → Creating a 2D projection of a 3D-scene-representation
  - * Rendering according to camera intrinsics + extrinsics
    - ↳ focal length, principal point
    - ↳ 6 DOF

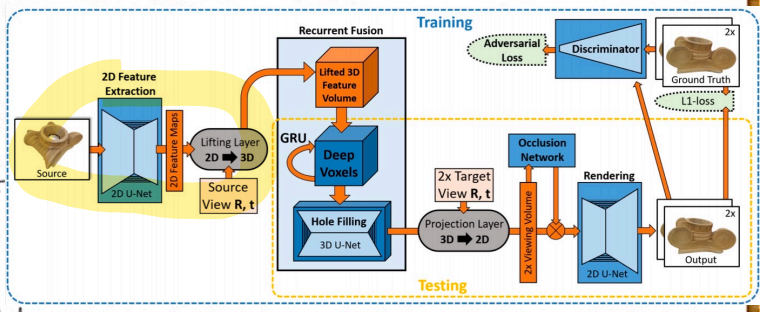


  - * Photorealistic renderings of "perfect" models are possible → Reconstructing 3D models from 2D views is difficult

- What is novel viewpoint synthesis?
  - * NN encodes the entire scene description (+lighting, materials, etc.)
  - * Input is camera pose + viewing direction → Output is novel view image

- Neural Rendering w/ Pix2Pix: (2017)
  - At train time: Pose + ground truth image → Reconstruction loss to train
  - At test time: Unseen pose → extrapolated, unseen image
  - * Problem: Pix2Pix uses conv. → 2D-operators need to learn 3D scene representation
    - ⇒ Artefacts, flickering

- → w/ Deep Voxels: (2019)
  - * Extracting features from 2D images and lift them to 3D voxel grid (according to camera pose R,t)
  - * Deep voxels represented as 3D U-net
  - * Projection from 3D to 2D according to target camera pose R,t
  - * Occlusion network masks out occluded voxels (→ depth estimation network)
  - ⇒ All operations are differentiable

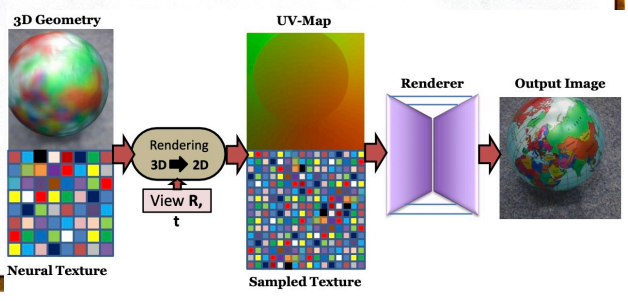


But limited resolution due to dense 3D voxel grid

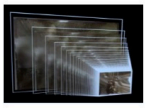
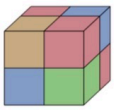
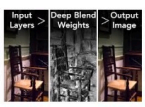
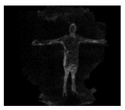
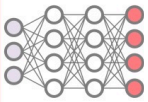
- Lifting from 2D to 3D works great
  - – No need to take specific care for temp. coherency!
- But: limited resolution due to dense 3D voxel grid

- → w/ an implicit representation:
  - ↳ Scene representation networks: (2019)
    - * Can generalize!
    - * Scene representation is ReLU MLP
    - * Renderer is generalized learned sphere tracing
      - ↳ How far until ray hits surface?
  - ↳ NeRF: Neural Radiance Fields: (2020)
    - * Scene repr. is ReLU MLP + pos. encoding
    - * Renderer is volumetric stratified sampling
    - * No generalization → MLP just memorizes to color of points in space ⇒ then interpolation

- → w/ Neural Textures: Features on a 3D mesh (2019)
  - * Map a neural texture on a given mesh (instead of RGB-texture; features), that live ontop of mesh)
    - "texels"
  - * Project 3D mesh + neural texture to 2D according to camera pose
  - * 2D U-net renders the realistic image



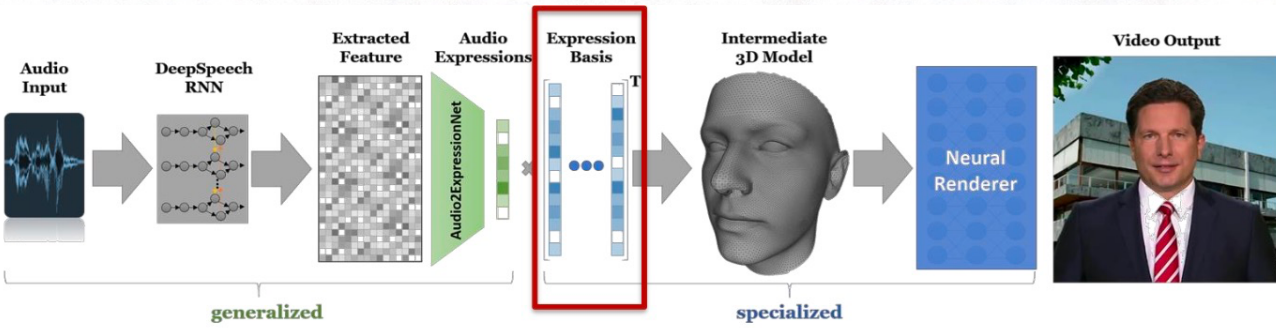
- Comparison of neural rendering approaches:

	Multi-Plane Images	Voxelgrids	Image-based	Point Clouds	Implicit Function
Scene Representation					
Renderer	(Alpha) compositing	Volumetric Ray-based	Rasterization	Splatting	Sphere-Tracing Volumetric
Pros	Fast rendering High quality Generalizes	"True 3D" High quality	High quality	High quality	High quality May generalize!
Cons	Only 2.5D Size	No reconstruction priors Memory $O(n^3)$	Requires good SFM No compact representation	Requires good SFM	Expensive rendering, training

- Neural voice puppetry:

only trained once
* Use the DeepSpeech model to extract features (logits of alphabet)
* Convert to expressions

trained per face
* ↳ drive facial model



Person-specific
Blendshape Expression Model

ADL4CV 9 : Deep Learning in higher dimensions

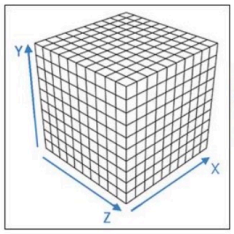
- Multi-dim. ConvNets:
  - 1D : Audio, Speech → e.g. WaveNet
  - 2D : Images
  - 3D : Videos, 3D data → 3D-shape/scene classification
    ⇒ requires more memory, but less data (as the viewing direction does not need to be learned)
  - 4D : dyn. 3D data, simulations

- Volumetric grids:

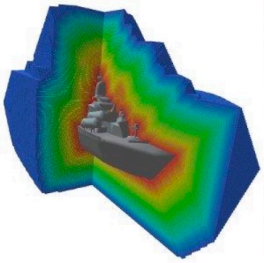
"voxel" ≙ pixel in 3D

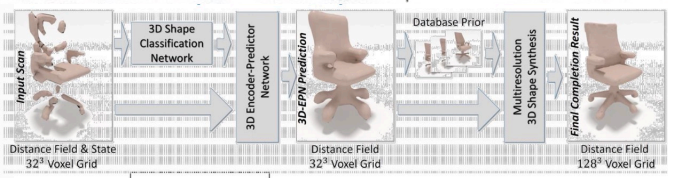
→ Volumetric data structures:
- Occupancy grids (binary value per voxel) — surface occupancy get smaller w/ higher resolution
- Ternary grids (is it empty space? surface point? unknown?)
- Distance field (every voxel encodes distance to closest surface)
  ↳ at surface point = 0
- Signed distance field (sign encodes if voxel is in front or behind surface)
  → empty space
  → unknown space

a field is a continuous representation

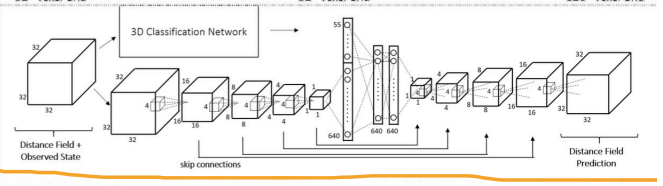


(binary) Voxel Grid

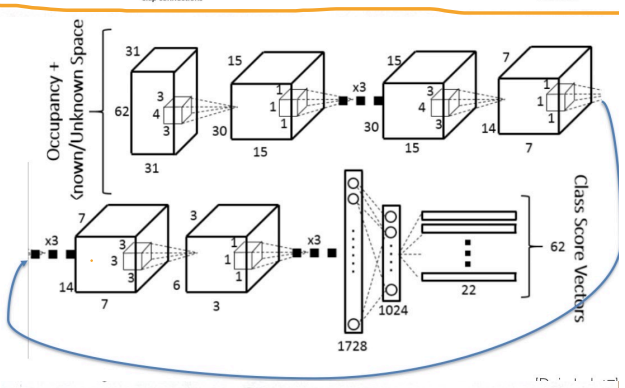




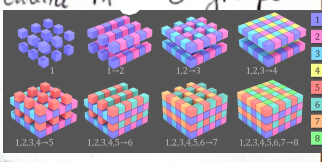
→ 3D shape completion:
- 3D U-net architecture
- Input is signed distance field
- Output is distance field
  ⇒ network makes prediction



→ Semantic Segmentation in 3D:
- Process scene volume in chunks (here 62 × 31 × 31)
- Input is signed distance field chunk
- Output are classification scores for the 62 central voxels
  ⇒ Sliding window / chunk — since input grid can vary in size
  ⇒ Doing shape completion while semantic segmentation helps the segmentation

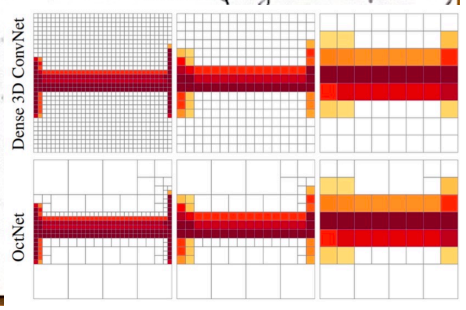


- Also w/ 3D autoregressive neural networks ⇒ 6×6×4 chunk in 8 groups



→ Volumetric grids need a lot of memory + processing time
→ sliding window

- Volumetric Hierarchies: → increase resolution of voxels close to a surface
  - Requires the 3D structure to be known beforehand → only for discriminative tasks (e.g. classification)
  - Voxels in (signed) distance field
  - Reduced memory consumption (and runtime)
    - → DenseNet: $O(n^3)$
    - → OctNet: $O(n^2)$
    - small performance hit
  - In generative models the model needs to predict whether to split voxels

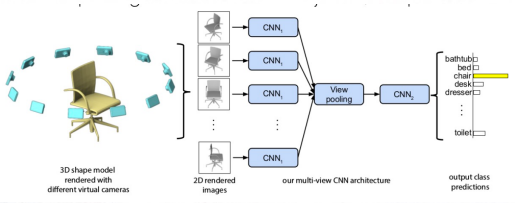


- Multi-view:

do not use 3D information explicitly → instead they use 2D rendering and aggregate features in the 2D domain

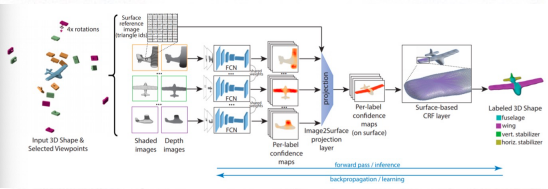
→ Classification:

* RGB imgs from fixed views around object
* view pooling (invariant w.r.t. # of views)
* Then classification



→ Segmentation:

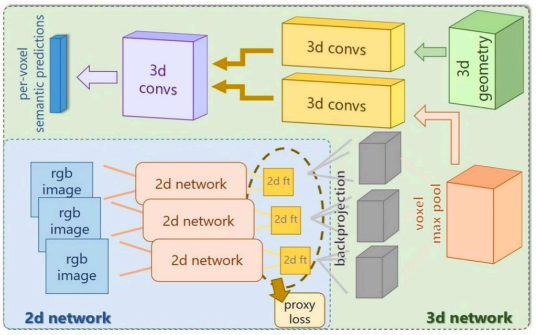
* 2D-segmentation on every view
  → output are per-label confidence maps
* Backprojection onto the object surface



- Hybrid: Volumetric + Multi-view:

* Extracts features from multi-view 2D imgs
* Backproject color features to voxel grid
* Add the color voxel grid to the geometry voxel grid

⇒ Color and geometry complement each other



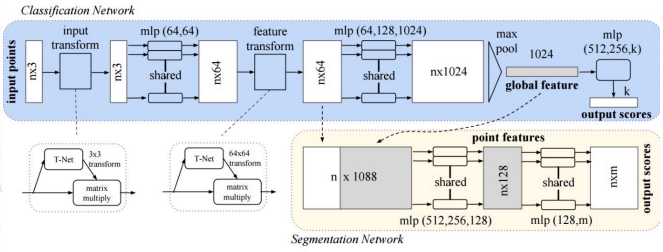
- Point Clouds: → Alternative to volumetric voxel grids

→ PointNet: (2017)

* n input points $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$
* extract per-point 1024-feature vector
* Most indicative features from MaxPool
  ⇒ output: class scores
* ⇒ concat per-point features with global features for segmentation scores

train & test is very fast

But: performance mostly worse than volumetric



PointNet++: (2017)

* Adresses sampling mismatch (when many samples are in groups far apart)

- Mesh-based: → Polygons represent 3D surfaces

* Need some differential geometry approx. and perform convolutions in this space

prone to noise on real data ⇒ No good results yet

- Sparse Convolutions: → Solve problem where dense 3D convs are expensive in memory and have the dilution problem (set of non-zeros grows quickly)

"Local convolution in the regions where we actually have a surface"

* Sparse Convs save previously active pixels in a hash-table
  ⇒ all others are masked out
  ⇒ less memory usage / allows for larger resolution
* Submanifold convolution

slower than dense conv due to lookup in hash-table

but: higher accuracy