CMSC740
Advanced Computer Graphics

Fall 2025

Matthias Zwicker

Background terminology

The course assumes you are familiar with the following terms. If you are not, you should read up on these via Wikipedia, chatbots, etc.

- (Artificial) neuron
  - Weights, bias, activation function
- Neural network architectures
  - Layers, hidden layers
  - Convolutional neural networks
- Normalization
  - Batch normalization
- Supervised learning
  - Loss functions, including L2 loss, L1 loss
  - Training data
- Stochastic gradient descent
  - Automatic differentiation/backpropagation
  - Adam

Deep learning/AI in graphics

- What problems can we address?
- What is special about deep learning? Why is it useful compared to other techniques?
- Limitations of deep learning/AI techniques?

Problems we'd like to address with deep learning

In computer graphics/vision
- Image/video -> class label
- Image/video -> set of 2D regions containing objects
- Image/video -> text description
- Image/video -> 3D geometry
- Low quality image -> high quality image
- Text description -> 3D object
- Text description -> video

Others
- Text in one language -> Translation to other language
- Text question -> answer
- Information about an environment (e.g. sensor data) -> best action of an agent to reach a goal
- Etc.

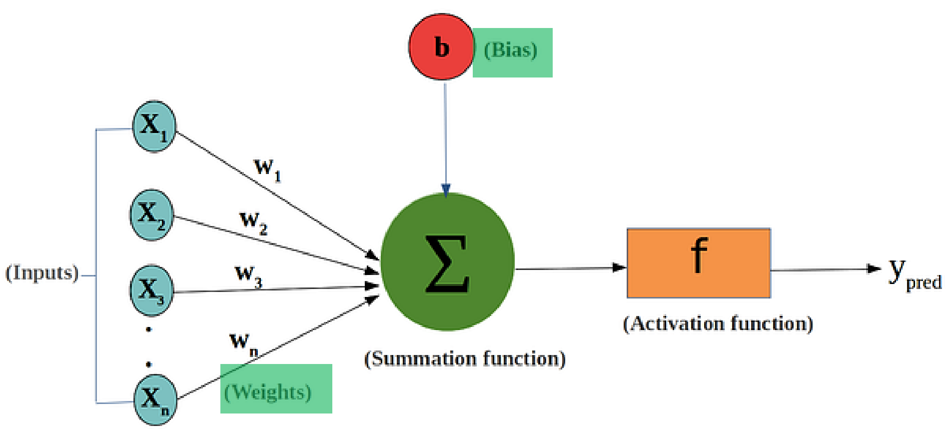
Abstractly: find mapping from complex, high-dimensional inputs to complex, high-dimensional outputs

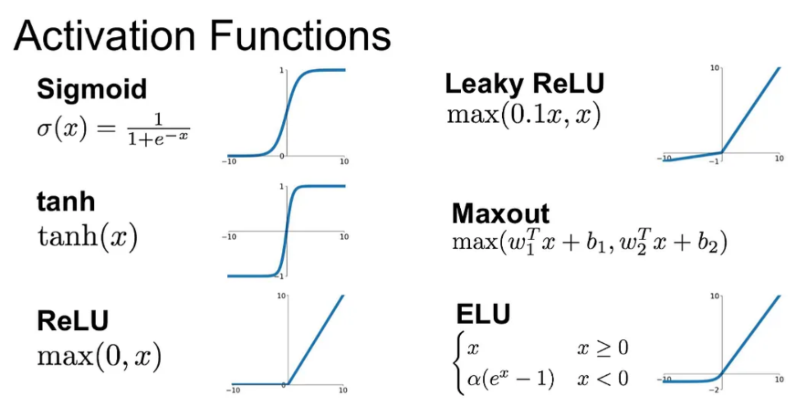
Deep neural networks

- Powerful type of non-linear functions
  - Specific function given by (potentially huge, billions) number of parameters (weights, biases)
  - Can approximate any input-output relationship by finding suitable parameters (universal approximation theorm) https://en.wikipedia.org/wiki/Universal_approximation_theorem
  - Work with high-dimensional inputs, outputs
  - Work with “brute force”, raw data representations (e.g., image/video as long vector of pixel values)
- Alternative types of non-linear functions?

Deep neural networks

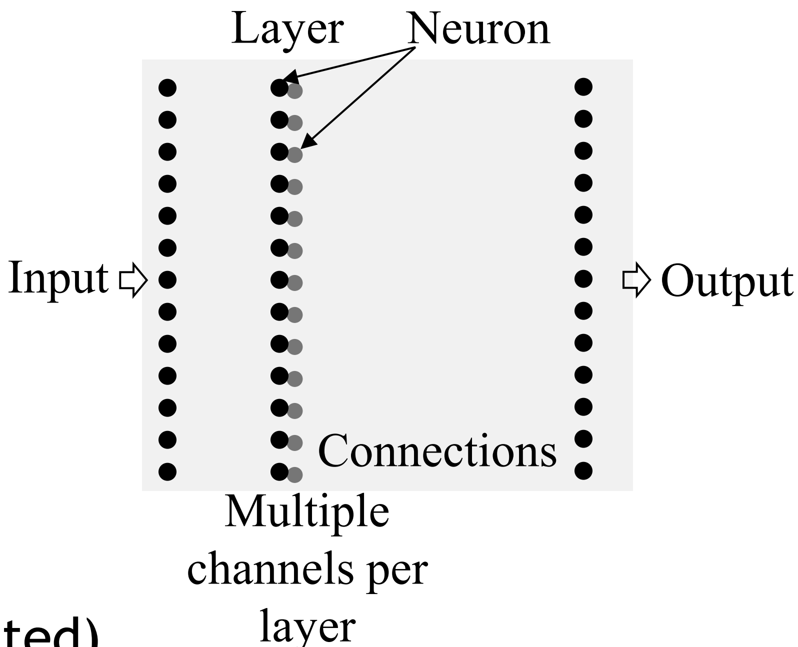
- Basic computational element: artificial neurons
  - Inputs x_i, output y_{pred}
  - Defined by weights w_i, bias b, activation function f
  - Parameters to be optimized: weights, biases





Deep neural networks

- Organized in layers
  - Thousands to millions of neurons per layer
  - Dozens to hundreds of layers (interior layers called hidden layers)
  - Billions to trillions of connections (weights)
  - Connections between layers, within layers, or across layers (skipping layers inbetween)



- Architectures (how neurons are connected)
  - Multilayer perceptrons (MLPs)
  - Convolutional neural networks (CNNs)
  - Residual networks
  - Transformers, multi-head attention
  - Etc.

Supervised learning

- Training data: set of corresponding pairs of desired inputs and outputs (e.g., image, text description, etc.)
- Objective: Find mapping (function) that fits/approximates training data; i.e., map each input to corresponding known output in training data
  - Minimize scalar loss function to quantify quality of fit/approximation
- Questions/challenges?
  - Collecting training data
  - Mathematical form of mapping, loss function
  - How to represent input, output
  - How to find mapping function that achieves objective (minimizes loss)

Stochastic gradient descent

- Simple algorithm to find function parameters (weights, biases), given training data and loss function
  - Loss function: quantify difference between training data (desired outputs) and network outputs as scalar number
- Iteratively
  - Compute gradient of loss function wrt. function parameters (weights, biases), using random (stochastic) subset (batch) of training data
  - Gradient: vector of same length as parameters, gives small change for each parameter that will reduce loss the most
  - Update parameters using gradient
- Advantages/disadvantages/alternatives?

Backpropagation

- Algorithm to compute gradient of neural network computations using automatic differentiation
- Automatic differentiation: algorithm for automatically computing derivatives of arbitrary concatenation of elementary functions (with known derivatives) using chain rule
  - Implemented using software libraries that maintain data structures, intermediate results to calculate chain rule “in the background”, without user requiring to add much code

(Mostly) open questions

- Generalization: how to characterize behavior of mapping function on inputs not seen in training data?
- Robustness: how to ensure that small changes in inputs or training data preserve accuracy of mapping function?
- Architecture design: given training data, how to characterize network architectures that will work well (accuracy on training data, generalization, robustness)?

Recap

- Rendering equation
- Monte Carlo path tracing
- (Multiple) importance sampling
  - Bidirectional path tracing
- Advanced appearance models
  - BRDFs/BSDFs/BSSRDFs to model specific materials
- Participating media

The good

- Simulate light transport in a physically accurate (in relation to human perception) manner
- Include complex material appearance models, complex geometry
- Generate photorealistic images (with sufficient modeling effort and render time)

https://corona-renderer.com/gallery

The bad

- Noise/variance, slow convergence, $O(1/\mathrm{sqrt}(n))$

55 sec, 32 spp, 12 threads

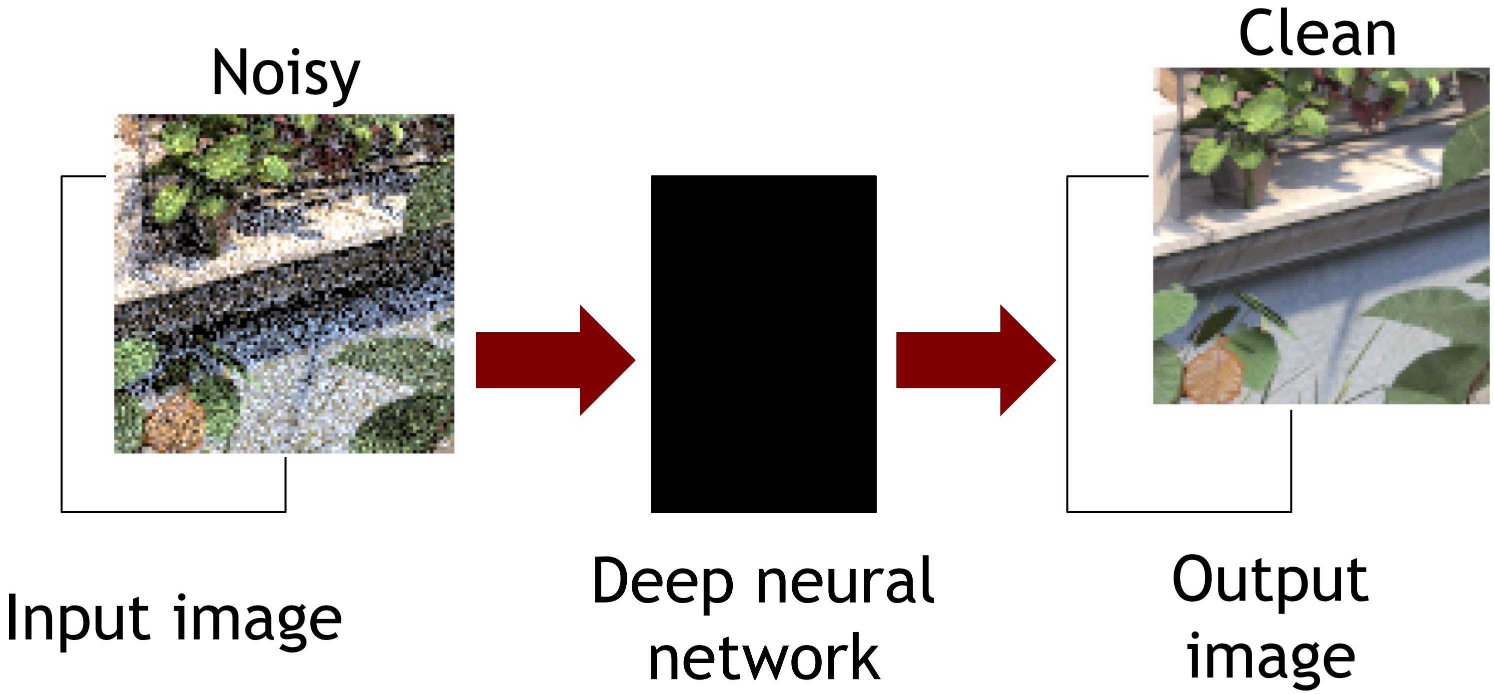
100 sec, 128 spp, 12 threads

http://cgg.unibe.ch/data/PG2013/

The solution?

The solution?

Image-to-image translation

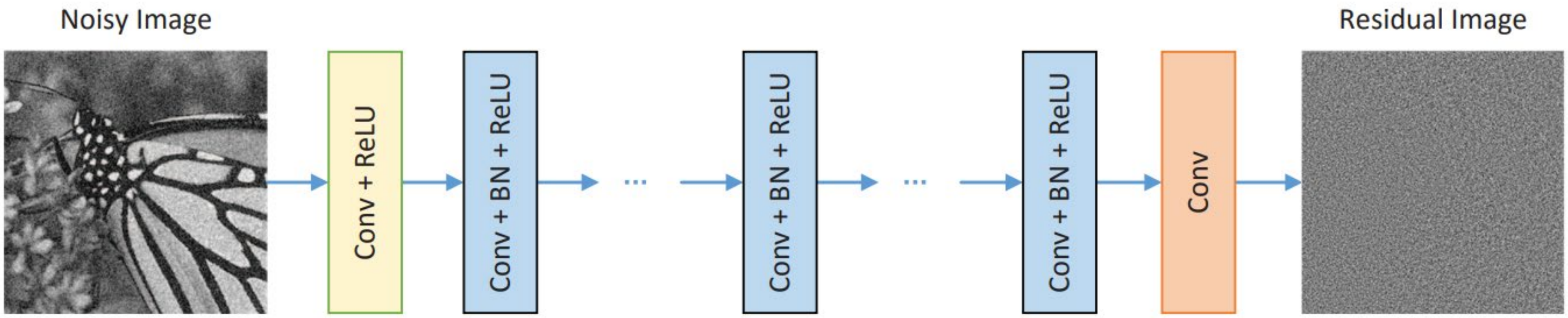


Various applications see https://phillipi.github.io/pix2pix/

Neural network architectures suitable for image-to-image translation?

DnCNN: straightforward denoising with deep CNNs

- “Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising”, Zhang et al., 2016
- Training data: pairs of noisy/clean images (add known noise to clean images)
  https://github.com/cszn/DnCNN
- 20 layers, 64 3x3 convolution filters in each layer (64 channels per layer)
- Batch normalization (additional computation to scale, shift outputs of each layer to zero mean, unit variance in each training batch)
- Predict residual (known noise) instead of pixel value
- **L2 loss:** for each pixel, square of difference between ground truth and network output; sum over all pixels, (no adversarial, L1, or feature loss)
- No downsampling/upsampling, no skip connections
- Train/optimize weights, biases on multiple noise levels

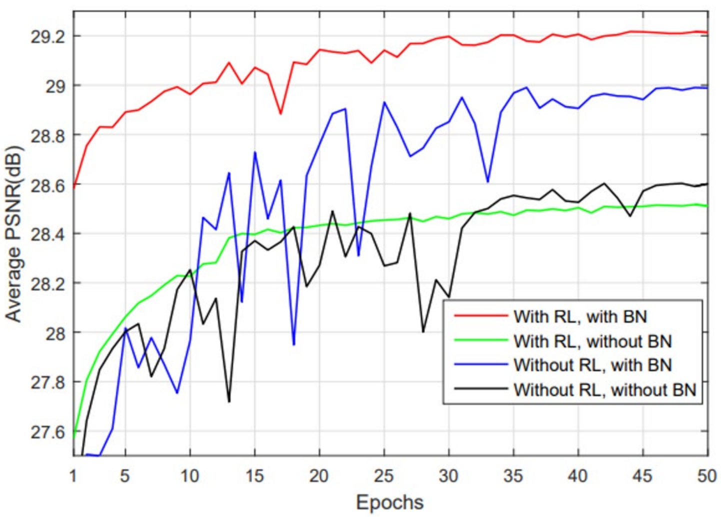


Conv: convolutional layer; BN: batch normalization; ReLU: rectified linear activation fct.

DnCNN

- Ablation study (comparison among different design decisions)
- Epoch: number of passes through entire training data set during gradient descent

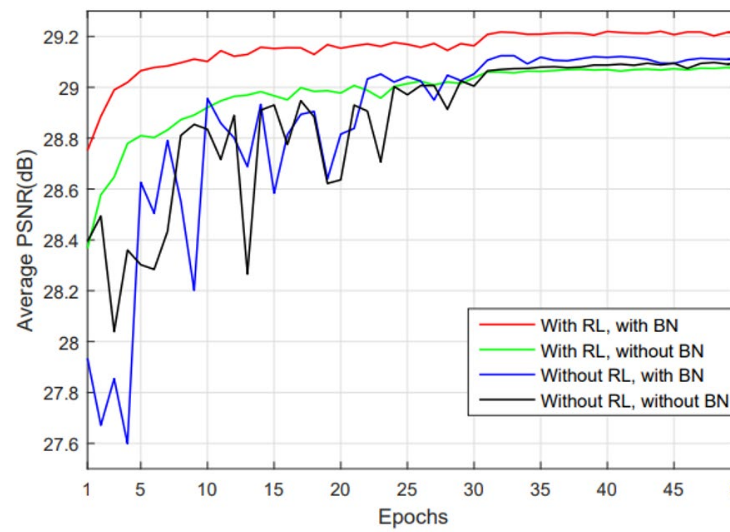
Accuracy



(a) SGD

Training progress

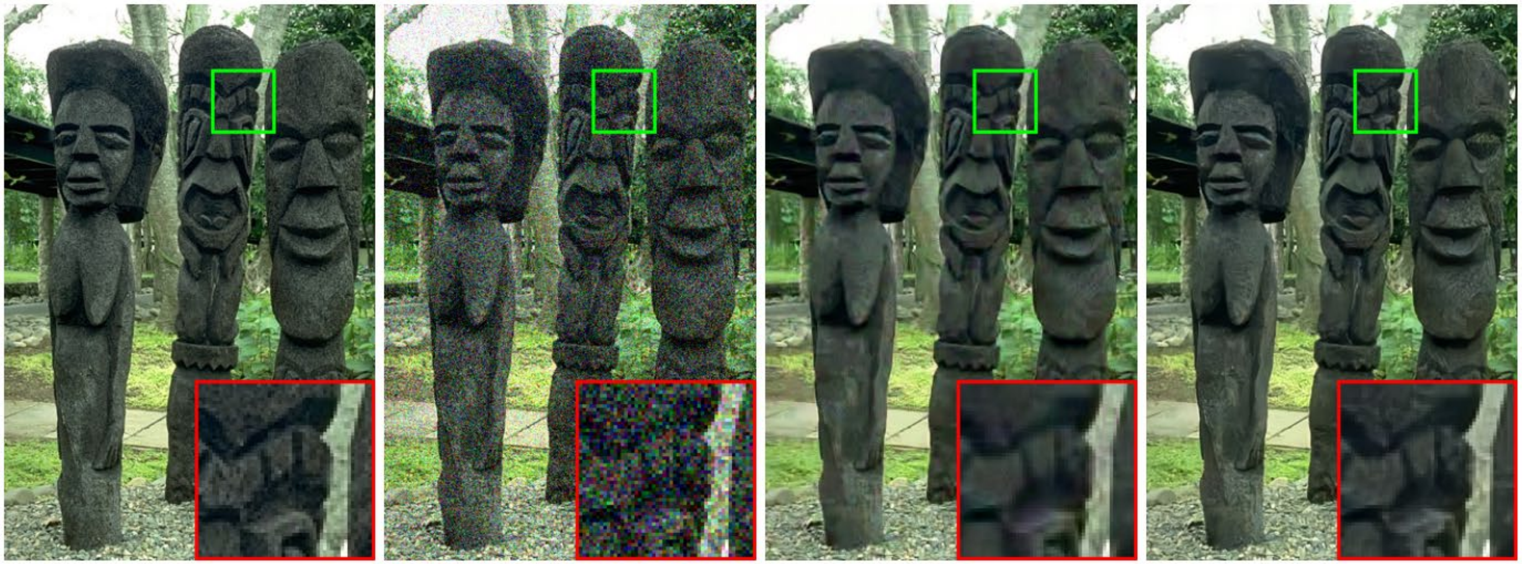
Stochastic gradient descent



(b) Adam

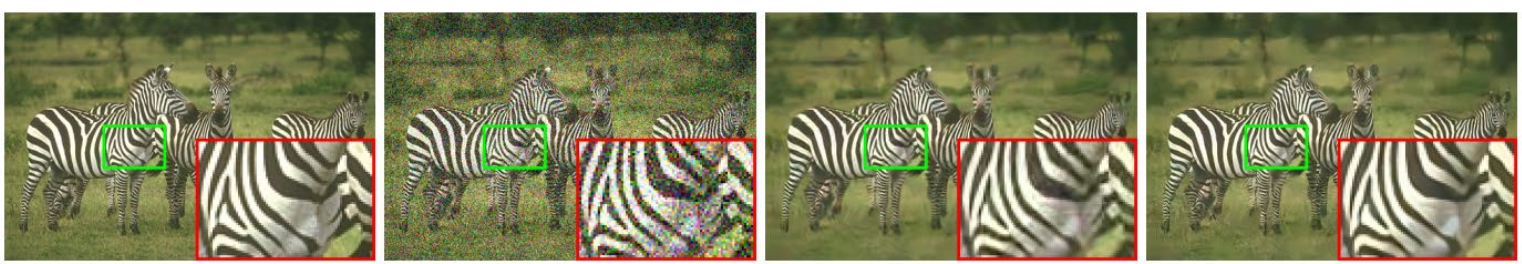
Adam (adaptive moment estimation) optimizer, improved version of SGD

DnCNN



(a) Ground-truth (b) Noisy / 17.25dB (c) CBM3D / 25.93dB (d) CDnCNN-B / 26.58dB

Fig. 6. Color image denoising results of one image from the DSD68 dataset with noise level 35.



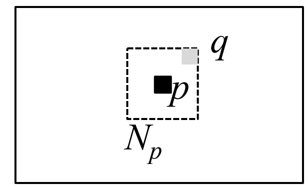
(a) Ground-truth (b) Noisy / 15.07dB (c) CBM3D / 26.97dB (d) CDnCNN-B / 27.87dB

Fig. 7. Color image denoising results of one image from the DSD68 dataset with noise level 45.

Denoising, generic approach

- Noisy pixel values c_p at each pixel p
- Denoising: replace each pixel with weighted average over neighor pixels c_q

$$\hat{c}_p = \sum_{q \in N(p)} w_{pq} c_q$$



  - Weights should be $0<w<1$, add up to one, (square) neighborhood N_p
- Assuming pixels c_q are independent random variables
  - Variance

$$V\left[\sum_{q \in N(p)} w_{pq} c_q\right] = \sum_{q \in N(p)} \left(w_{pq}^2 V\left[c_q\right]\right)$$

Variance reduction

$$\sum_{q \in N(p)} w_{pq}^2 << 1$$

  - Expected value, if c_q are i.i.d (independent, identically distributed, https://en.wikipedia.org/wiki/Independent_and_identically_distributed_random_variables)

$$E\left[\sum_{q \in N(p)} w_{pq} c_q\right] = \left(\sum_{q \in N(p)} w_{pq}\right) E\left[c_p\right] = E\left[c_p\right], \quad \text{where i.i.d implies} \quad E\left[c_p\right] = E\left[c_q\right]$$

Denoising, classical approach

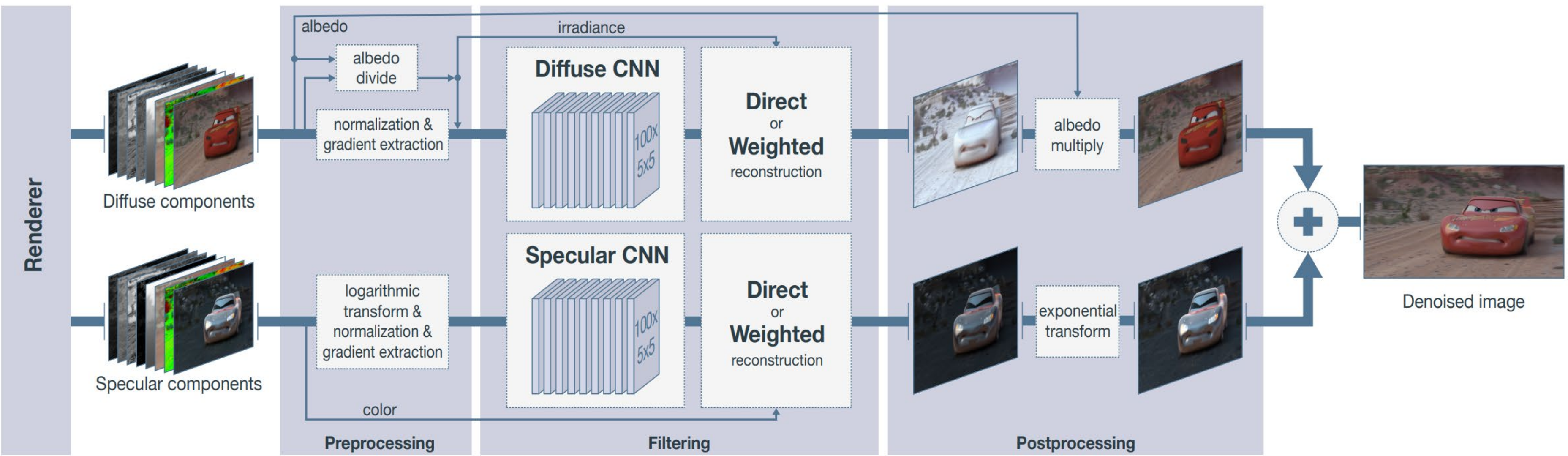
- Example: box filter, $w_{pq} = 1/|\mathrm{N}(p)|$
  - Variance reduction by $1/|\mathrm{N}(p)|$
  - Larger filter (larger size of neighborhood $|\mathrm{N}(p)|$) leads to more variance reduction
  - But: bias (pixels are not i.i.d in practice, i.e. $\mathrm{E}[c_p] \neq \mathrm{E}[\mathrm{c}_q]$, hence box filter changes expected value of pixel, in general), causes blur
- Goal: techniques to determine filter weights in range $0<w<1$, such that bias is avoided
- Ideas?
  - Make w_{pq} high if expected pixel values $\mathrm{E}[c_p]$ and $\mathrm{E}[\mathrm{c}_q]$ are similar, otherwise, make w_{pq} low

Denoising using kernel prediction

- "Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings", Bako et al., 2017
- Predict weights w_{pq} ("kernel") in neighborhood of each pixel using neural network
- Other ideas to improve performance of denoising in the case of Monte Carlo rendered images?

CNN to denoise Monte Carlo renderings

"Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings", Bako et al., 2017



Any per-pixel data produced by renderer that may be useful for denoising (depth, normals, BRDF parameters)

Bako et al., SIGGRAPH 2017

http://cvc.ucsb.edu/graphics/Papers/SIGGRAPH2017_KPCN/

Specialties (vs usual denoising CNNs)

- Take advantage of additional information (besides RGB) produced by renderer
  - Additional per-pixel input channels (normal, depth, BRDF parameters) stacked together with RGB colors
- Diffuse/specular separation of input
  - Diffuse: albedo divide
  - Specular: color-luminance separation, log-transform

Specialties (vs usual denoising CNNs)

Denoising kernel prediction

- Network output at each pixel p is k x k vector $\mathbf{z}^L{}_p$
- Weight for pixel q in filter of pixel p

$$w_{pq} = \frac{\exp([\mathbf{z}_p^L]_q)}{\sum_{q' \in \mathcal{N}(p)} \exp([\mathbf{z}_p^L]_{q'})},$$

Ensure weight normalization (sum to 1)

- Filtered output at pixel p

$$\widehat{\mathbf{c}}_p = g_{\text{weighted}}(\mathbf{X}_p; \boldsymbol{\theta}) = \sum_{q \in \mathcal{N}(p)} \mathbf{c}_q w_{pq}.$$

- Advantage: can prove faster training convergence in simplified case; better generalization in practice

Network, training & test data

Network

- 8 hidden layers, 100 convolution kernels (channels) of 5×5 in each layer
- L1 loss (for each pixel, absolute difference between network output and ground truth, sum over all pixels)

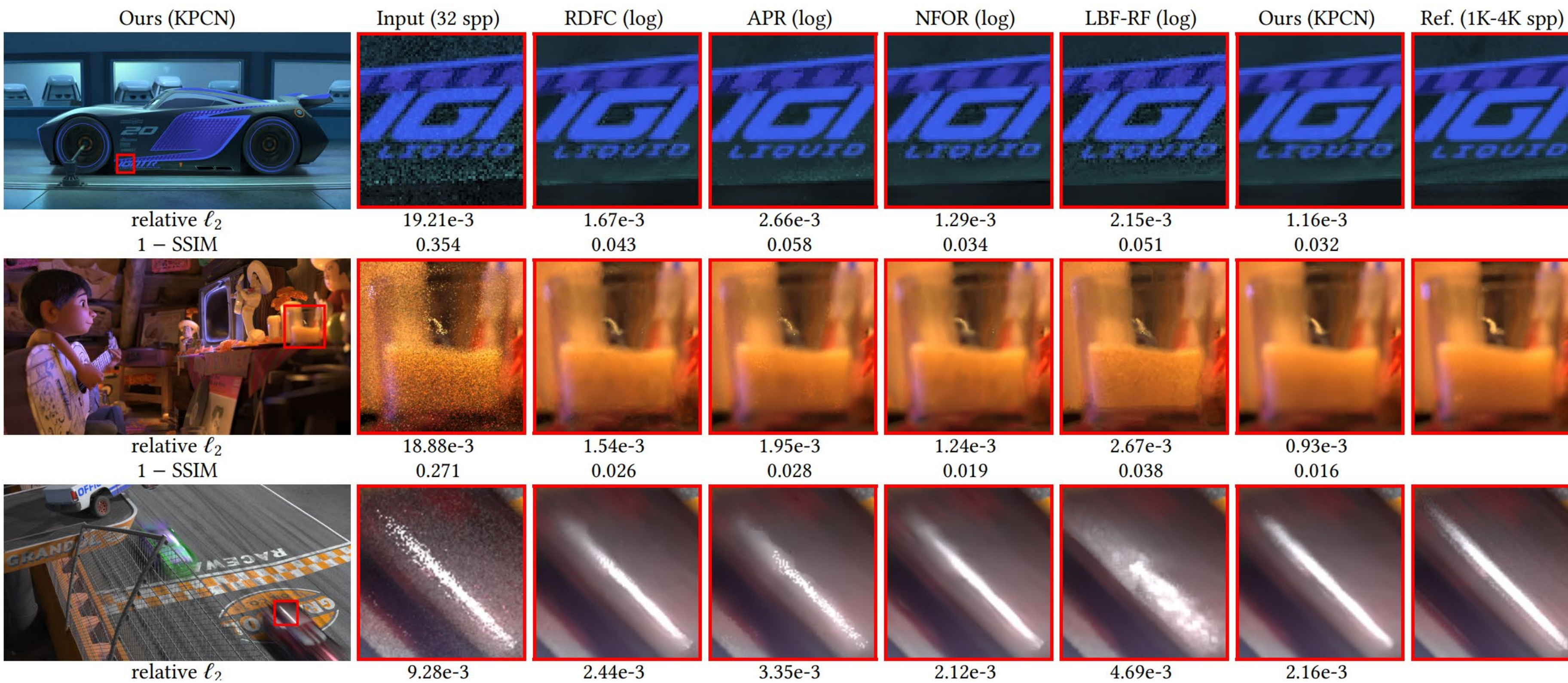
Training data

- 600 frames from movie “Finding Dori”
  - Noisy: 32 spp
  - Reference: 1024 spp

Test data

- 25 frames from other movies
  - Mmotion blur, depth of field, glossy reflections, and global illumination

Results



Bako et al., SIGGRAPH 2017

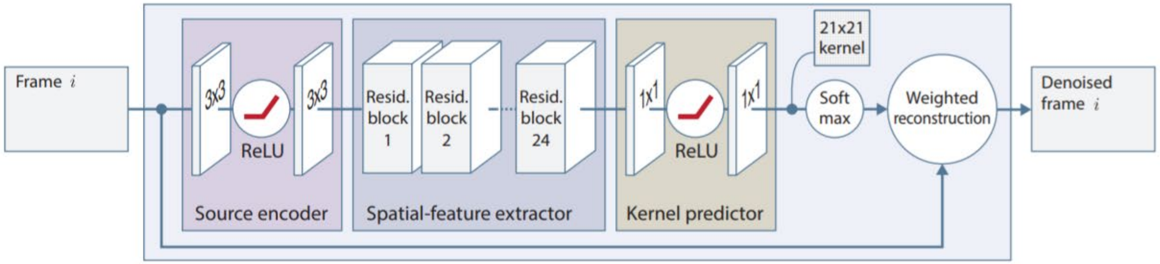
http://cvc.ucsb.edu/graphics/Papers/SIGGRAPH2017_KPCN/

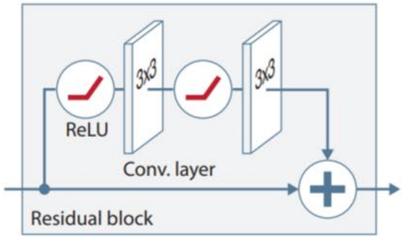
Denoising animations

- Frame-by-frame denoising leads to temporal flickering in animations
- Need to filter in 3D space-time
- How to implement this with CNNs?

Improvement of Bako et al. 2017

- “Denoising with Kernel Prediction and Asymmetric Loss Functions”, Vogels et al., ACM Siggraph 2018
  - Residual blocks (avoid vanishing gradient problem, https://en.wikipedia.org/wiki/Vanishing_gradient_problem)
  - Modular training (source encoder, spatial-feature extractor, kernel predictor) allows specialization to different renderers with less training data



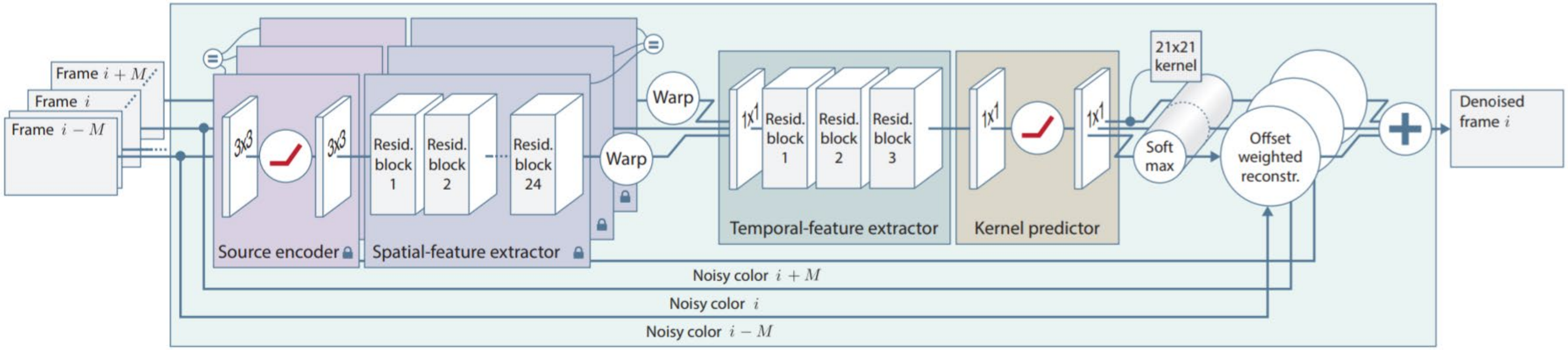


Input of res. block added to its output, forming “shortcut”

http://zurich.disneyresearch.com/~fabricer/publications/vogels-2018-kpal.pdf

Temporal denoising

- Input $2M+1$ frames simultaneously
- Use source encoder and spatial-feature extractor modules pre-trained on single frames
- Warp frames to align with reference i using motion vectors or optical flow
- Predict single $21\text{x}21\text{x}(2M+1)$ kernel to denoise frame i



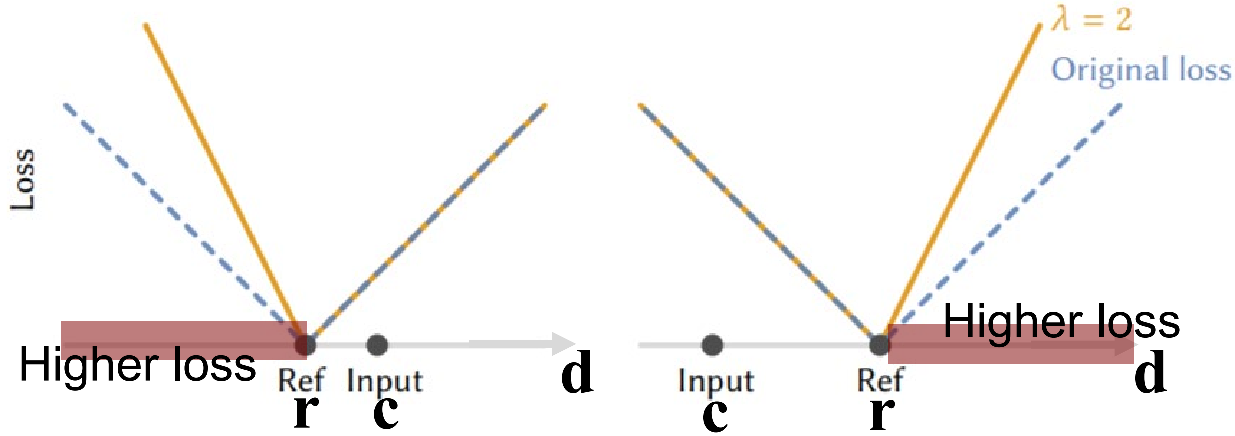
http://zurich.disneyresearch.com/~fabricer/publications/vogels-2018-kpal.pdf

Asymmetric loss

- Loss function l'_λ
  - Ground truth $\mathbf{r}$, network output $\mathbf{d}$, noisy color $\mathbf{c}$, Heaviside step function H, user parameter λ

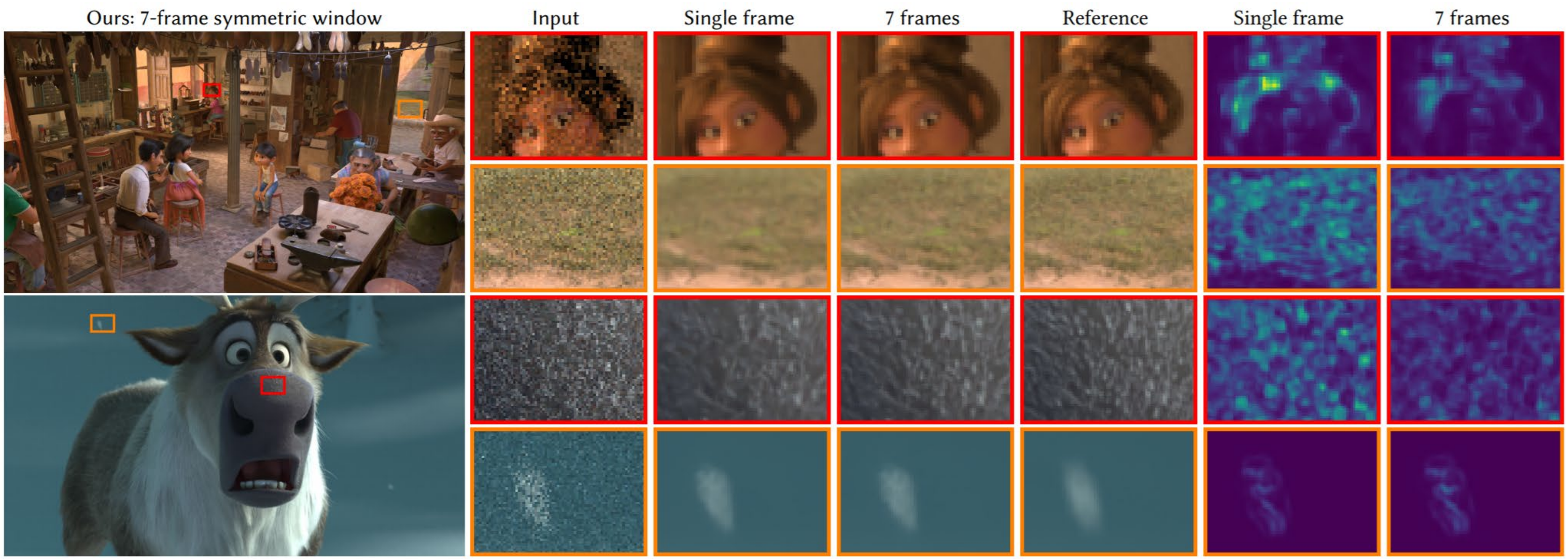
$$\ell'_\lambda(\mathbf{d}, \mathbf{r}, \mathbf{c}) = \ell(\mathbf{d}, \mathbf{r}) \cdot (1 + (\lambda - 1)H((\mathbf{d} - \mathbf{r})(\mathbf{r} - \mathbf{c})))$$

  - "Penalize network outputs $\mathbf{d}$ more if they are on the other side of the reference than the noisy input $\mathbf{c}$"
- User can adjust λ to control bias/variance tradeoff



http://zurich.disneyresearch.com/~fabricer/publications/vogels-2018-kpal.pdf

Results



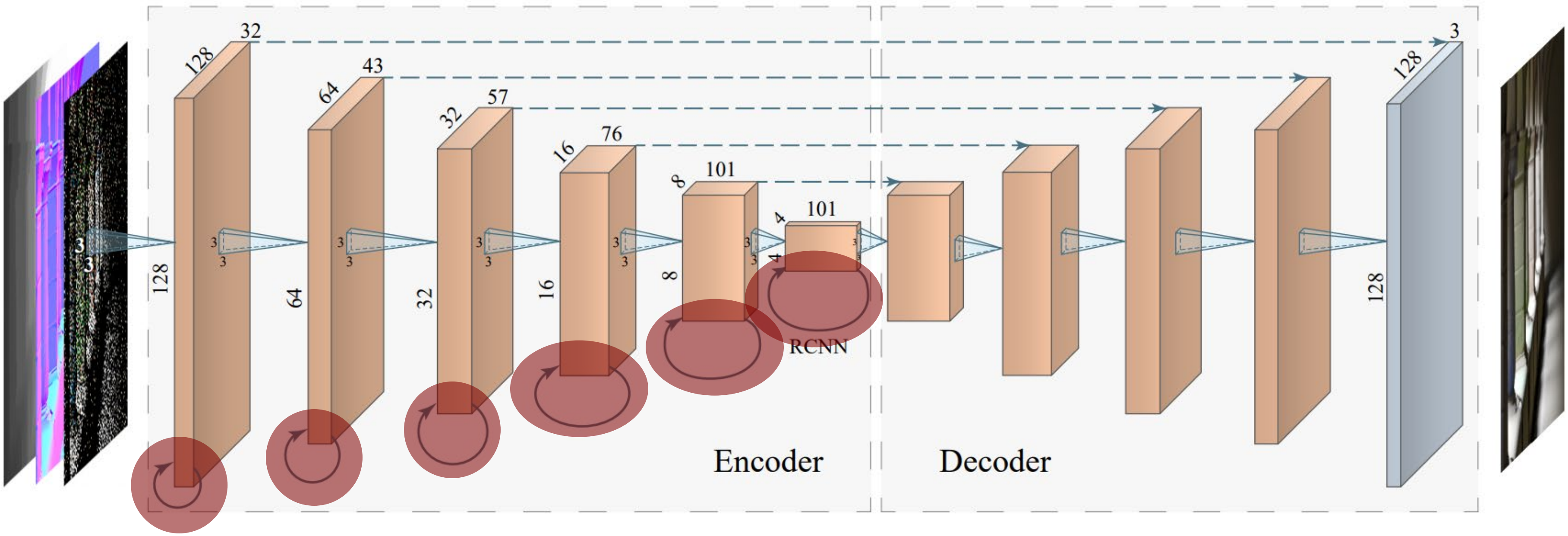
http://zurich.disneyresearch.com/~fabricer/publications/vogels-2018-kpal.pdf

Interactive, GPU rendering

- Two similar products from Nvidia using recurrent convolutional auto-encoder
  - Auto-encoder: network that consists of encoder and decoder parts
- Denoising for ray tracing built into Nvidia Optix
  https://developer.nvidia.com/optix-denoiser
  - Optimized for inference performance, milliseconds per frame
- Nvidia DLSS
  https://www.nvidia.com/en-us/geforce/technologies/dlss/

Recurrent neural network RNN

- Chaitanya et al, Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder, SIGGRAPH 2017
  http://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf



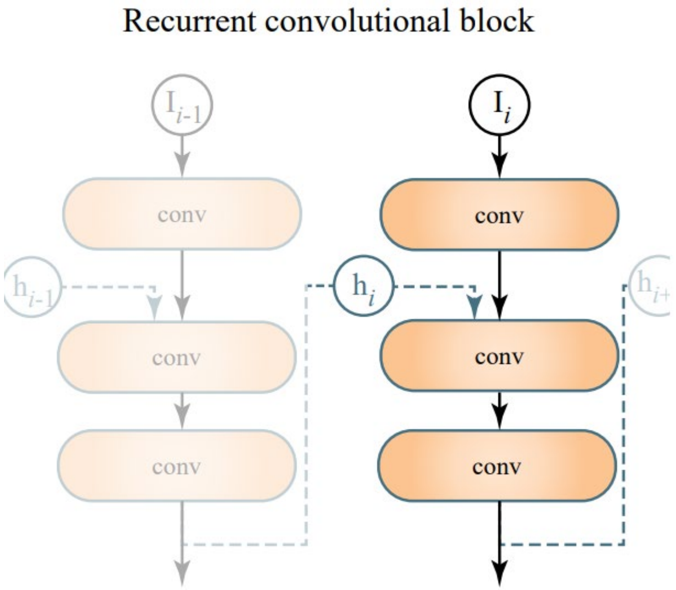
Output of previous frame used as input of next frame

Specialties

- Input to network
  - RGB colors
  - “G-buffer” with view-space shading normals (2D vectors), depth, material roughness
  - Divided by diffuse albedo (remove texture complexity from input)
- Loss
  - Spatial: L1, L1 on spatial gradients
  - Temporal: L1 on temporal gradients

Training

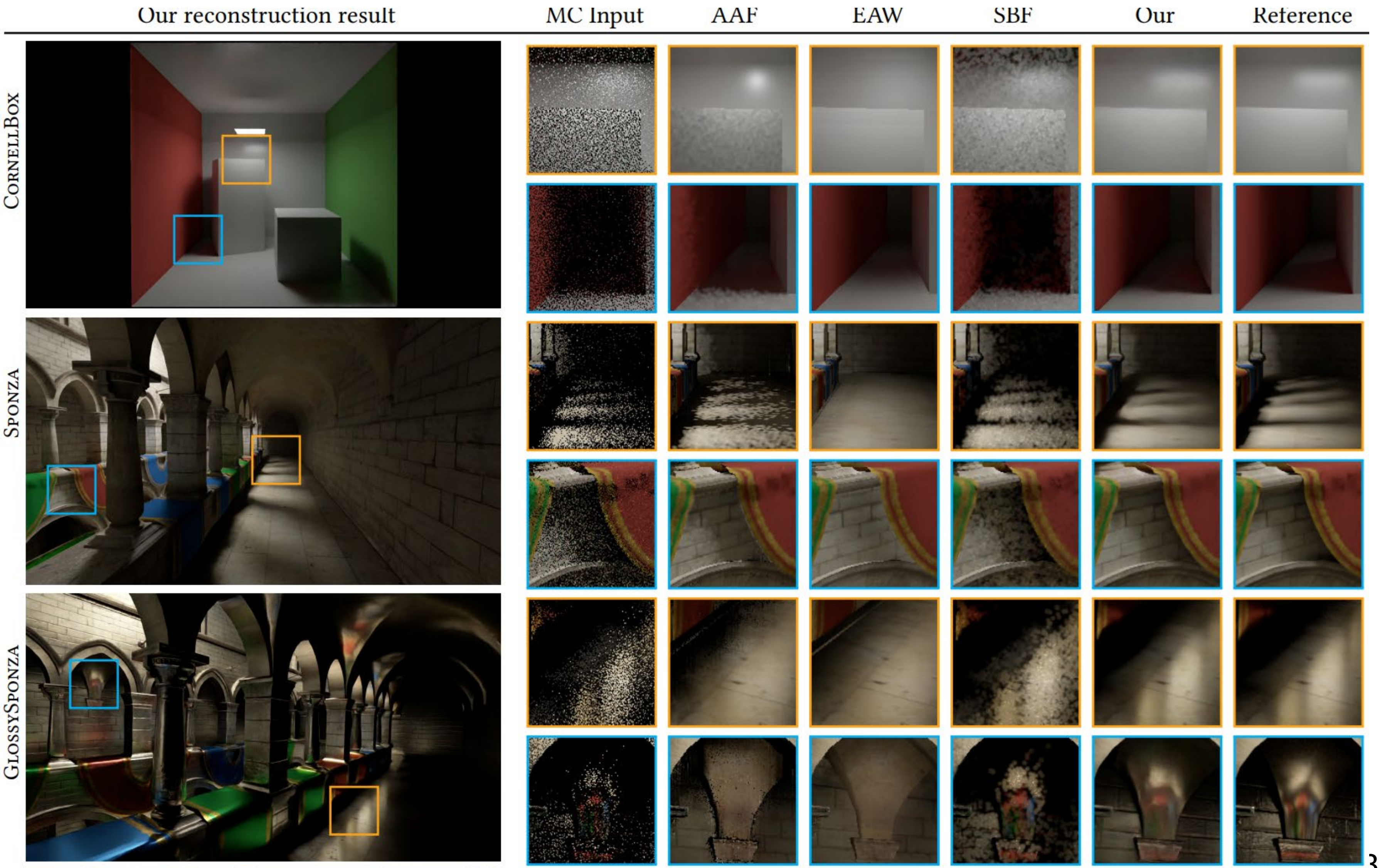
- "Back propagation through time"
  https://en.wikipedia.org/wiki/Backpropagation_through_time
  - Replicate feed forward parts to unroll recurrent connections
- "Adam" SGD solver
  https://arxiv.org/pdf/1412.6980.pdf
  - Gradients are noisy in SGD
  - Compute moving average over SGD steps



Two replicated feed forward steps for frame i-1, i, recurrent connection from i-1 to i

Results (inputs with 1spp)

http://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf



Results

- Video
  http://research.nvidia.com/publication/interactive-reconstruction-monte-carlo-image-sequences-using-recurrent-denoising

Computation times

- 54.9ms for 1280×720 pixels on NVIDIA (Pascal) Titan X
- OptiX-based path tracer from 70ms to 260ms in SanMiguel for 1 spp
  https://developer.nvidia.com/optix

Deep learning for rendering

- What else could we learn to improve rendering efficiency?
  - Keep more data from the renderer than 2D images (e.g., operate on 4D images (light fields), for denoising, interpolation)
  - Use image-to-image translation to add expensive light transport effects (global illumination, subsurface scattering, motion blur, …)
  - Learn the entire rendering function

    scene description -> neural network -> image with global illumination

Learn to importance sample

- "Learning to Importance Sample in Primary Sample Space", Zheng, Zwicker, 2018
https://arxiv.org/abs/1808.07840
- "Neural Importance Sampling", Muller et al. 2018
https://arxiv.org/abs/1808.03856