1 Intro

Perceptron: while $\exists x.w^T x > 0 \neq y$ do $w = w + \eta(y - \widehat{y})x$
Note: If the data is separable, the algorithm converges in finite time

MLP: $\forall l \quad x^{(l)} = \sigma((w^{(l)})^T x^{(l-1)} + b^{(l)}), f(x; w, b) = x^{(L)}$
Activation function: If the activation function is linear, than the MLP is an affine transformation with a perciluar parameterization.

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$, $\nabla : \sigma(x) \cdot (1 - \sigma(x))$

Softmax: $softmax(x_i) = \exp(x_i)/\sum_j \exp(x_j)$
Requirements: Output must be positive
output must be between $[0, 1]$
sum of all outputs must be 1, e.g. $\sum_i^M softmax(x_i) = 1$

tanh: $\nabla : 1 - tanh(x)^2$

MLE: maximize $\log L(\theta) = \log \prod p(x_i|\theta) = \sum \log p(x_i|\theta)$
Note: Least squares and CE are ML estimators
Algorithm:
Write down probability distribution
Decompose into per sample probability
Minimize negative log likelihood

L_{CE}: $-\frac{1}{N} \sum y_i \log(\sigma(w^T x_i)) + (1 - y_i) \log(1 - \sigma(w^T x_i))$ (MLE)
Note: Cross-entropy loss is a maximum likelihood estimator
Assumption: y's are Bernoulli $Ber(\sigma(w^T x))$ distributed.
From there we maximise the weights over the probability:
$P(D|w) = \prod^N \sigma(w^T x_i)^{y_i}(1 - \sigma(w^T x_i))^{1-y_i}$.
Taking the negative log-likelihood results in the given loss.

Universal Approximation: $\exists g(x) = \sum v_i \sigma(w_i^T x + b_i) \approx f(x)$ and $|g(x) - f(x)| < \epsilon$, σ non-const, bounded, continus
Intuition: We can create bumps with only two hidden neurons, having a lot of neurons can create many bumps to approximate any function.
Note: $\sigma : \mathbb{R} \to \mathbb{R}$ must be non-constant, bounded and continous
Note: $f \in [0, 1]^m$, e.g. must be in the m-dimensional hypercube

SGD: $\theta = \theta - \eta \nabla C(\theta)$, **Batch:** gradient avg. of whole dataset
SGD has more fluctuations and will never reach the minimum, but dances around it. The convergence is usually faster though, since it updates the parameters more often
Batch is doing one update per epoch, inefficient if there are a lot of samples in the dataset

2 Convolutional Neural Network

Generalization: trade-off specifity vs. invariance
We need to trade-off specifity and invariance (created by affine transformations of an object and different lightings) to increase the generalization ability of the model.

Receptive Field: area triggering neuron, inhibit or excit
Note: Areas can be on the Retina, Skin, Tounge, etc.
From Hubel and Wiesels experiments we learned that stimulus covering the whole cat retina, most neurons didn't fire, since the excitatory and inhibitory stimulus canceled each other out. The light must fall on specific regions to excite, forming specific excitement *patterns*.
Direction and alignment of the light affected the firing, and was used to figure out the alignment of the receptive field.

Hierarchy: **S**imple cells, specific stimuly, **C**omplex cells, complex stimuly
Note: Size of receptive fields tends to get larger, the more complex the cell becomes.
Sensory receptors connected to cell in the brain form the receptive field of that cell. This receptive field can be very noisy in simple cells.
The connection of such simple cells to complex cells can form complex triggering patterns, f.e. a line in a certain direction. This hierarchy can be extended to hypercomplex shapes.

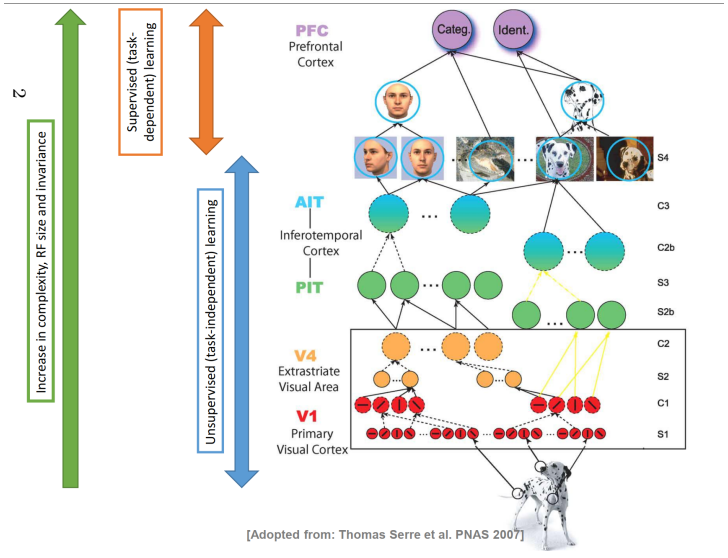


Figure 1: Complexity, unsupervised and supervised learning hierarchy

HMAX: S: $y = \exp(-\frac{1}{2\sigma^2}\sum^{n_{S_k}}(w_j - x_j)^2)$, C: $y = \max_{n_{C_k}} y_j$
The HMAX model is used to describe how our visual system works, it uses a hierarchical structure. x is the input to the cell, w is the weight of the cell, sigma is the size of the receptive field of a cell.
Note: Any linear, shift-equivariant transform can be written as convolution.
Linear: $T(\alpha u + \beta v) = \alpha T(u) + \beta T(v)$
Invariant: $T(f(u)) = T(u)$
Note: We want this in classification, b.c. a cat in the middle of the picture should still be classified as cat if it is shifted or roted
Equivariant: $T(f(u)) = f(T(u))$
Note: In Edge detection very important, if we shift the edge in the image, we also want the response to shift the same way
Correlation: $I'(i,j) = \sum_{m=-k}^{k}\sum_{n=-k}^{k} K(m,n)I(i+m, j+n)$
Note: I is the image, K is the Kernel, N(i,j) is the neighbourhood of a pixel.
Shift-invariance: The Kernel is usually parameterized as K(i,j,m,n), e.g. the weights of the Kernel depend on the location in the image. Removing i,j-dependence makes the kernel shift invariant.

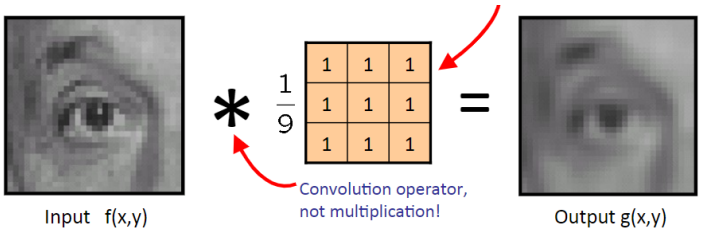


Figure 2: Linear filtering is applied with the convolution operator to compute a new image from the neigbourhood of each pixel of the old image

Conv: $I'(i,j) = \sum_{m=-k}^{k}\sum_{n=-k}^{k} K(m,n)I(i-m, j-n)$
Alternative representation would be
$I'(i,j) = \sum_{m=-k}^{k}\sum_{n=-k}^{k} K(-m,-n)I(i+m, j+n)$
Convolution acts as a point-spread function, what we want to learn are the weights of the kernel. It can be done via matrix multiplication.
Correlation = Convolution: iff $K(i,j) = K(-i,-j)$
Matrix $I * K$**:** band matrix $K \in \mathbb{R}^{n+m-1 \times n}$, k_i on diagonal

$$(I * K) = \begin{pmatrix} k_1 & 0 & \cdots & 0 \\ k_2 & k_1 & \ddots & \vdots \\ k_3 & k_2 & k_1 & \\ \vdots & k_3 & \ddots & \\ 0 & \vdots & 0 & k_m \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ \vdots \\ \vdots \\ I_n \end{pmatrix}$$

Figure 3: CNN Convolution as matrix

Diff.: Conv. with kernel [-1, 1], $\frac{\delta f}{\delta x} \approx \frac{f(x_{n+1},y) - f(x_n,y)}{\Delta x}$
Dimension: $\frac{I_{height} + 2 \cdot \text{padding} - \text{dilation} \cdot (K_{height} - 1) - 1}{\text{stride}} + 1$
Note: If we have 6 filters, the produced "new image" would be of size 28x28x6

Figure 4: Visualisation applying filters to an image

Weight sharing: Same feature detector K for whole I
Note: This makes the feature detector robust agains affine transformations, as it detects the feature in the whole image once trained
Stride: used to reduce size, replaces pooling layers
Dillation: fast increase of rec. field, get global context
CNN-fwd: $z_{i,j}^{(l)} = w^{(l)} * z^{(l-1)} + b = (\sum_{m,n} w_{m,n}^{(l)} z_{i-m,j-n}^{(l-1)}) + b$

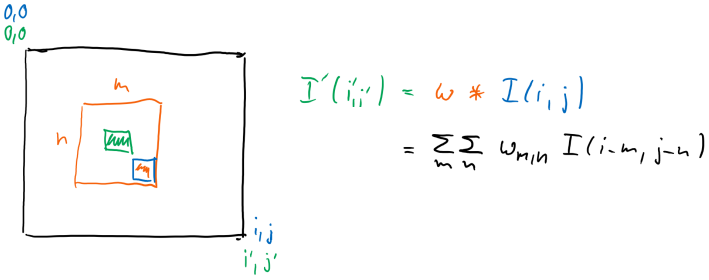


Figure 5: Visualisation forward pass

CNN-bwd (z): $\delta_{i,j}^{(l-1)} = \frac{\delta C}{\delta z_{i,j}^{(l-1)}} = \sum_{i',j'} \frac{\delta C}{\delta z_{i',j'}^{(l)}} \frac{\delta z_{i',j'}^{(l)}}{\delta z_{i,j}^{(l-1)}}$
$= \sum_{i',j'} \delta_{i',j'}^{(l)} w_{i'-i,j'-j}^{(l)} = \delta_z^l * ROT_{180}(w^{(l)})$
If we take the derivative of $\delta z_{i',j'}^{(l)}$, only certain weight terms stick, the rest is going to be zero.
Note: Backward path is just a convolution with the flipped kernel

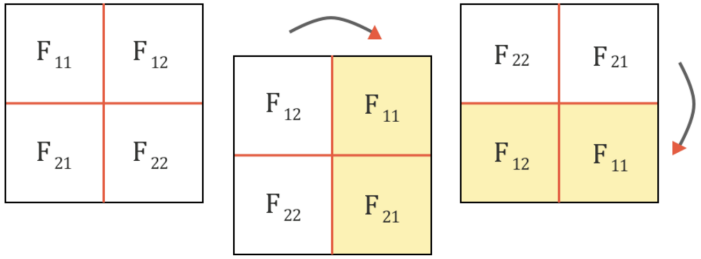


Figure 6: Flipped convolution operator

TODO: Do calculation yourself to see why it is flipped, or better, understand the relation to the "standart" convolution

Note: Each layer holds more complex "patterns", this resembles the neuron complexity hierarchy we have seen in nature.

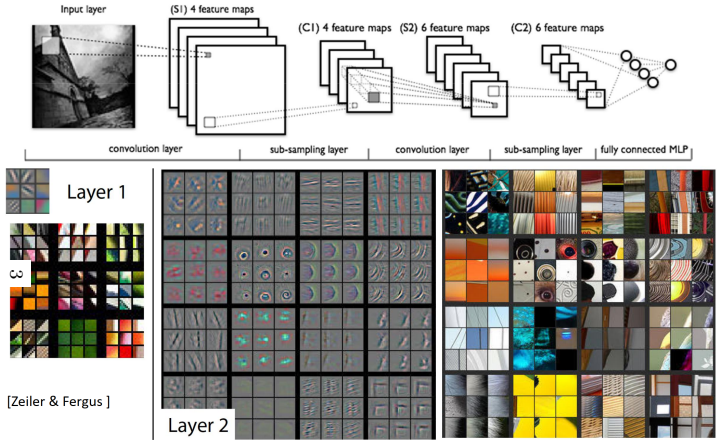


Figure 7: Visualisation of output of cnn layers

Pooling: $z^l = \max\{z_i^{l-1}\}$, $\frac{\partial z^l}{\partial z_i^{l-1}} = 1$ if was max else 0

Reduces size of each activation layer, downsampling. This helps in extending the activation region of downstream pixels, as we are putting more info in a smaller region. It also helps in reducing the noise in pictures.
There are other pooling strategies that can be applied, but with the MAX-pooling, we gain robustness to local changes. For example, if a digit rotates a bit, the region probably still has the same max pixel and thus is robust against rotations.

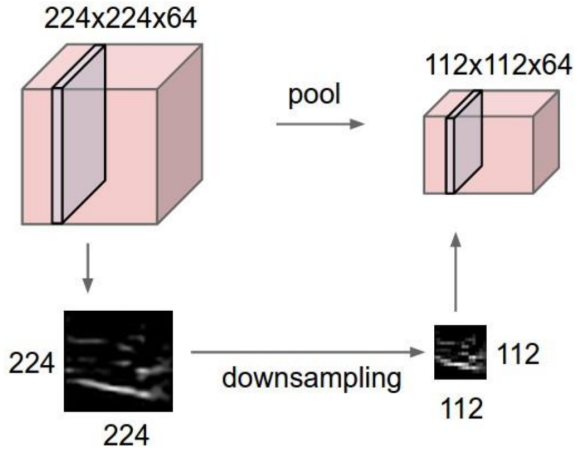


Figure 8: Visualisation of pooling layer

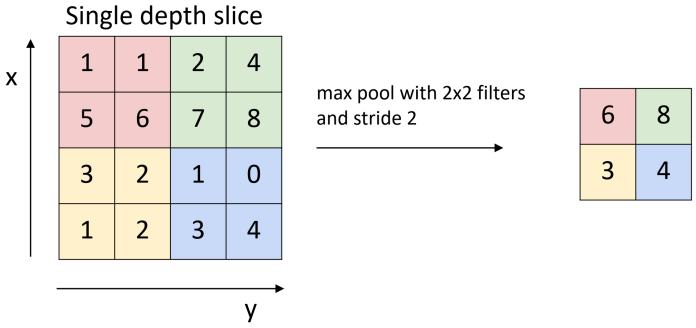


Figure 9: Visualisation of max pooling layer

2.1 — Evolution of architectures

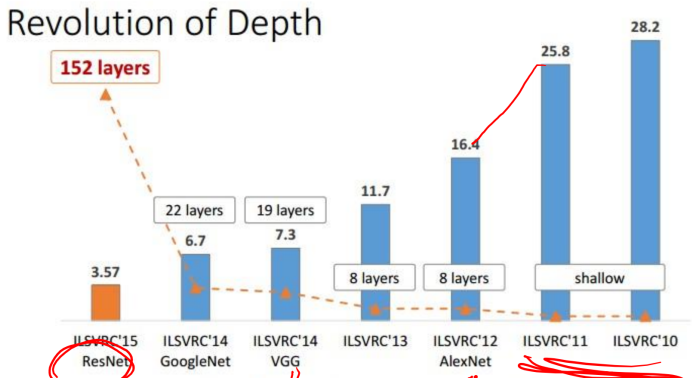


Figure 10: Evolution of NN architecture

2.2 — VGG vs. AlexNet

less kernels/filters (↓ params), more layers (↑ perceptive field)
Larger receptive field means that the net can respond to patterns that are larger spread apart.
Each large filter, f.e. 11x11, can be represented using multiple 3x3 filters. The number of parameters for one 11x11 filter is 121, which is equal to using 5 3x3 filters with 45 parameters.

2.3 — GoogleNet

More layers, removed fully connected layer on the top.
Inception: use 1x1 conv. layers to reduce layer depth
Because of small kernels and deep networks, the number of channels (depth) is going to be huge. Inception modules downsamples the activation maps, reducing the number of channels by convolution, and thus reducing the number of parameters.
Note: In addition, GoogLeNet uses auxiliary classification heads throughout the CNN, to make sure the gradients don't dry out.

2.4 — ResNet

Residual-Con: Skips weight layers with residual connections.
Note: A deep network should at least perform as good as a shallow one (set all additional layers to identity).
This approach lead to the believe that failing to perform is an optimisation issue, not a design issue.
ResNet one (drastic) downsampling and then residual layers. The "Ultra-deep" version is not used often, in general ResNet18 is used.

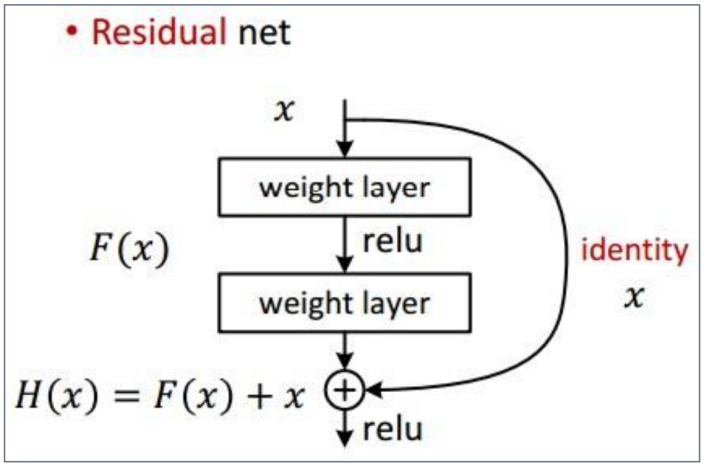


Figure 11: Evolution of NN architecture

3 Fully Convolutional Neural Network

The main difference to classical CNN is that we don't want to only work with fixed sized images as input. This limitation comes from the fully connected layers in the end.

Semantic segment.: extract patch, run through cnn, classify + size independen, - only local context We extract each patch and run it through a CNN. This let's us classify a pixel by it's local neighbourhood, but we loose the whole context of the pixel, as we are not looking at the whole image anymore.

Approach: learn features of the input image (encoding), which is followed by a decoder who projects the learned features to the higher resolution pixel space. We get rid of the fully connected layer at the end of the network.

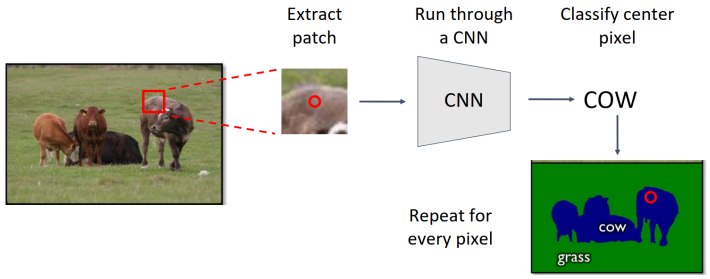


Figure 12: Pixel-wise classification pipeline

Downsample: pooling or strides, otherwise very expensive Downsampling helps with performance, as not the whole image is run through convolutions. But, it results in low resolution and fuzzy object boundaries. Newer architectures can mitigate this by copying the learned features from the downsampling stage into the upsampling stage.

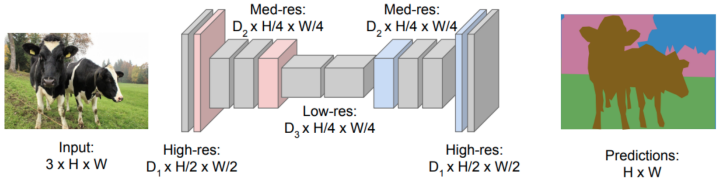


Figure 13: Downsample and than upsample to increase resolution again

Nearest Neighbor: copy value to the whole output Upsampling can be seen as interpolation, increasing the resolution of the signal.

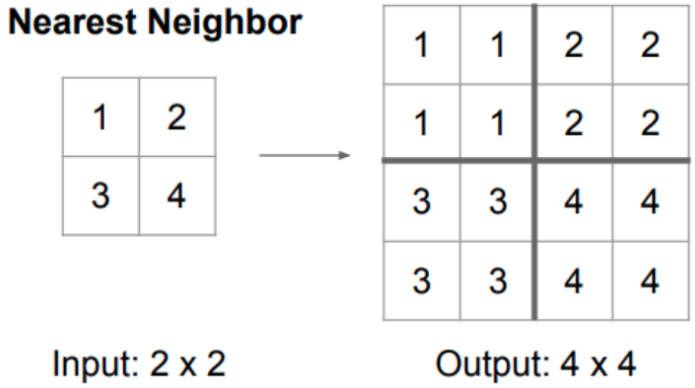


Figure 14: Copy input pixel to all output pixels

Bed-of-nails: zero all outputs but one copy of input

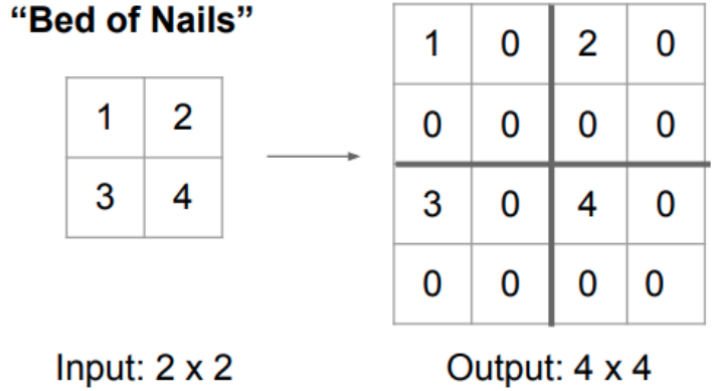


Figure 15: Copy input pixel to one output pixels, rest zero

Max-unpooling: remember max element from downsampling, use that in upsampling

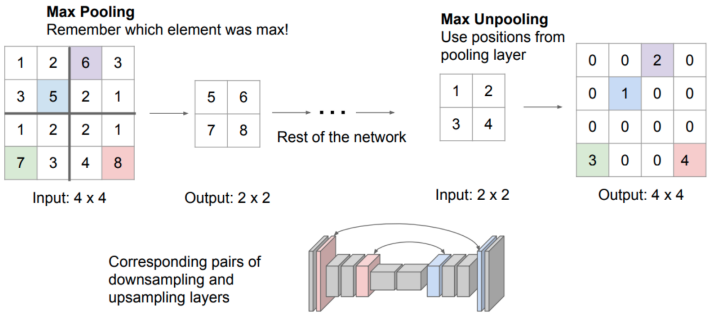


Figure 16: Copy input pixel to location of the max pooling pixel

Learnable Upsampling: input as weight, add up filters in output

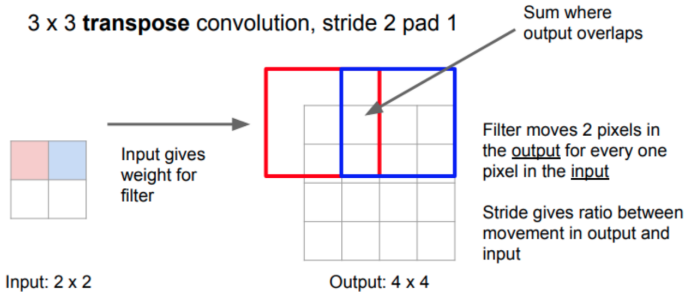


Figure 17: Input pixel gives the weight for a filter

UNet: copy early stage tensors to upsampling, combines local and global feature maps
Main idea: combine global and local feature maps by copying corresponding tensors from earlier stages into later stages.
Also, the downsampling information is pretty shallow, upsampling performance is improved when including "predownsamplin" information.

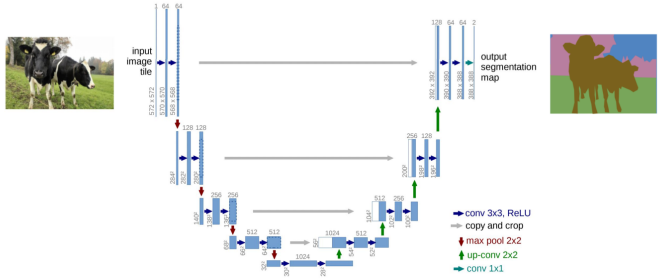


Figure 18: U-Net architecture

4 Recurrent Neural Network

The main advantage of RNN's over vanilla NN's or CNN's is the flexibility when it comes to input size.
NN has a fixed input/output size and computes a fixed amount of computational steps (defined by number of layers).
RNN can work over sequences of data, learning the transistion given the (implicitly stored) past data.

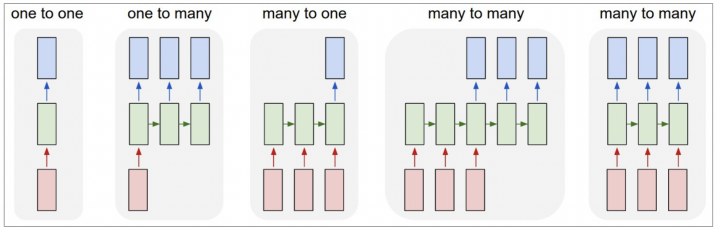


Figure 19: RNN 1. NN, 2. Image captioning, 3. Sentiment analysis, 4. Translation, 5. Video classification

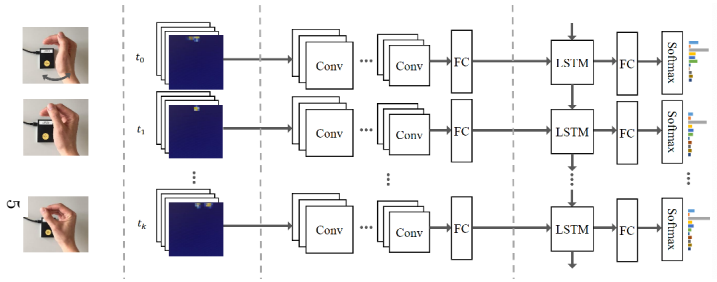


Figure 20: RNN Video classification architecture

RNN: $\hat{y} = W_h h^t$, $h^t = tanh(W[h^{t-1}, x^t])$,
$h \in \mathbb{R}^n, W \in \mathbb{R}^{[n \times 2n]}$ + variable sequence length
Goal: Learn transition function knowing what's worth keeping.
Tanh is important, otherwise we can't forget information.

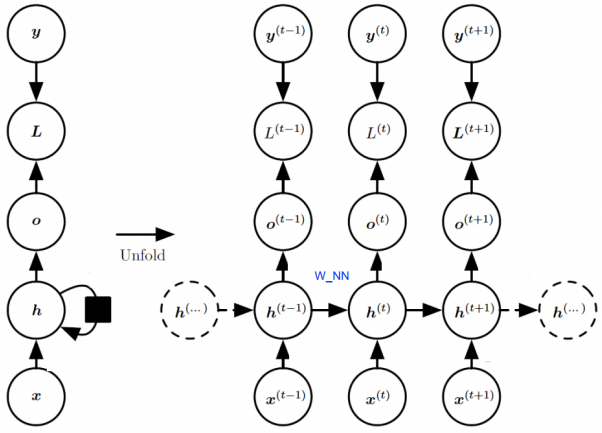


Figure 21: Visualisation of complex RNN

Backprop: $\frac{\delta L}{\delta W} = \sum_{t=1}^{S} \sum_{k=1}^{t} \frac{\delta L^t}{\delta y^t} \frac{\delta y^t}{\delta h^t} (\prod_{i=k+1}^{t} \frac{\delta h^i}{\delta h^{i-1}}) \frac{\delta^+ h^k}{\delta W}$
When backpropagating, we always have to consider all intermediate steps via the chain rule, as they recursively depend on W.

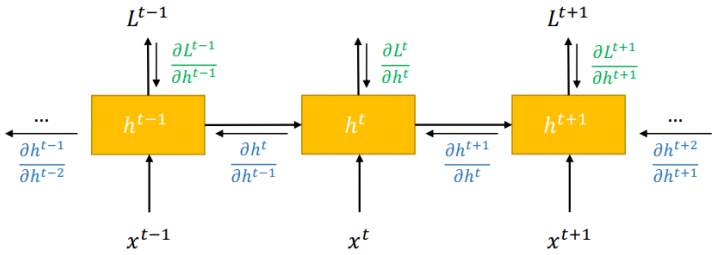


Figure 22: RNN Backprop unrolled

Gradient Problem: $h^t = W^T h^{t-1} = (W^T)^t h^1 = (Q^T \Lambda^t Q) h^1$
- explode: clip (stability), - vanish: memory cell
The Eigenvalues of the weigh matrix explode or vanish when the sequence length gets to big. Regularizing the Eigenvalues is proven to reduce the learning capabilities of the learning-system drastically, and is thus not an option
Exploding gradients: exploding gradients can be managed by clipping after a certain size
Vanishing gradients: Add memory cell

$$\frac{\partial L^t}{\partial W} = \sum_{k=1}^{t} \frac{\partial L^t}{\partial y^t} \frac{\partial y^t}{\partial h^t} \frac{\partial h^t}{\partial h^k} \frac{\partial^+ h^k}{\partial W}$$

we are interestedin this

$$\frac{\partial h^t}{\partial h^k} = \prod_{i=k+1}^{t} \frac{\partial h^i}{\partial h^{i-1}} = \prod_{i=k+1}^{t} W_{hh}^T diag[f'(h^{i-1})]$$

$$\forall_i, \left\| \frac{\partial h^i}{\partial h^{i-1}} \right\| \leq \left\| W_{hh}^T \right\| \left\| diag[f'(h^{i-1})] \right\| < \frac{1}{\gamma} \gamma < 1$$

$$\left\| \frac{\partial h^t}{\partial h^k} \right\| < (\eta)^{t-k}$$

Figure 23: Proof that vanishing Gradients are a problem

Naive Memory: $c^{(t)} = W_c c^{(t-1)} + W_g g^{(t)}$, $h^{(t)} = tanh(c^t)$
LSTM: $c_t^l = f \odot c_{t-1}^l + i \odot g$, $h_t = o \odot tanh(c_t^l)$,
$f, i, o, g = [\sigma, \sigma, \sigma, tanh] \odot W^l [h_{t-1}^l h_t^{l-1}]^T$, $W^l \in \mathbb{R}^{4n \times 2n}$
1) forget, 2) new info, 3) output gen
To mitigate the vanishing gradients, we introduce a long term memory unit. There are three phases to this: forgetting, new information and output generation.
1) Forgetting happens through manipulating c with $f \in [0, 1]$
2) New information is compiled by trying to figure out what to add/ remove to the current memory, as $g \in [-1, 1]$
3) Output generation is done by mixing the short-term input with the long-term memory

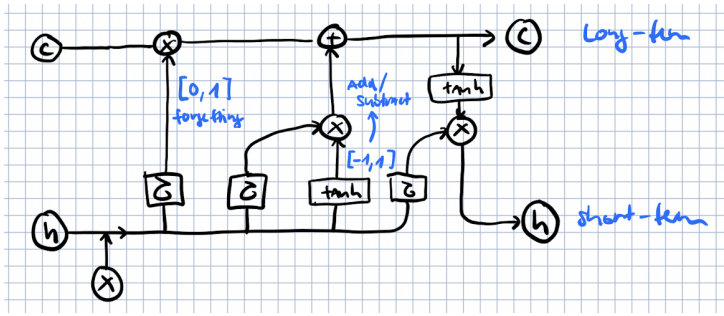


Figure 24: LSTM Visualization

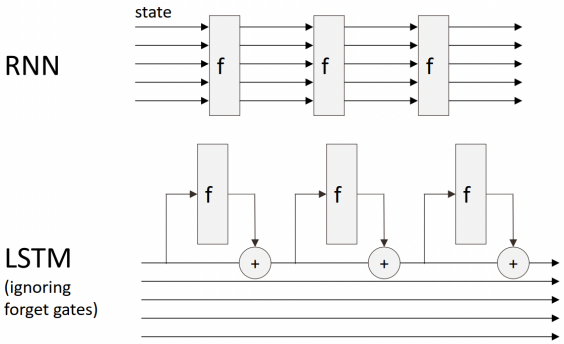


Figure 25: LSTM vs. RNN

Taxonomy of generative learning:
Supervised learning: Tries to learn a function that maps $X \Rightarrow Y$, used in classification, regression, object detection and segmentation.
Unsupervised Learning: The Goal here is to learn the hidden structure of the data, such as in clustering, feature learning, dimensionality reduction or density estimation.
Generative Modeling (our Goal): Given training data, learn a distribution and generate new samples drawn from the learned distribution. Learn p_{model} close to p_{data}, generate from p_{model}

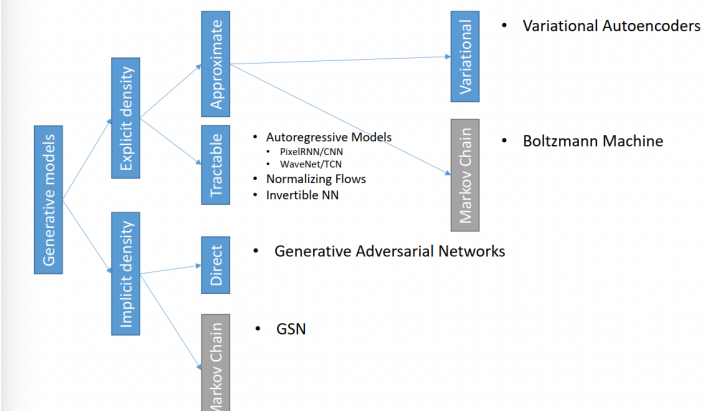
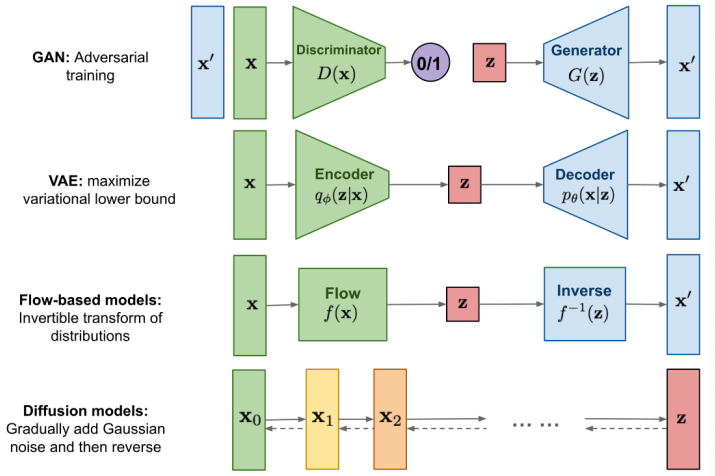


Figure 26: Taxonomy of variational models



5 Variational Auto Encoders

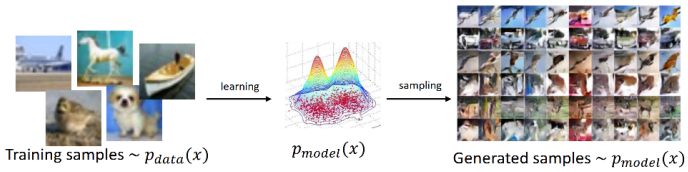


Figure 27: VAE Pipeline

AE: optimize $\theta_f, \theta_g = \arg\min \sum_n^N \|x_n - g_\theta(f_\theta(x_n))\|^2$
The encoder projects the original input to a latent space Z. It has to figure out what are the important features worth keeping (f.e. when dim(Z) < dim(X)).
The decoder learns to construct an output from a sample of Z.
In the optimal case, the encoder-decoder pipeline approximates the identity function of the data.

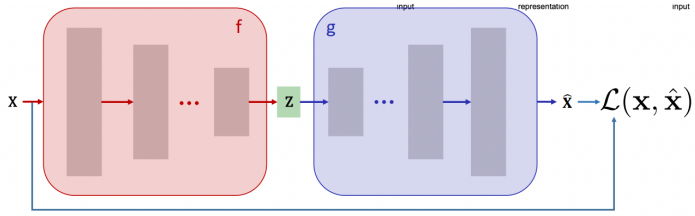


Figure 28: Auto-Encoders Model with loss

PCA: $z = f(x) = Wx + b$, $\hat{x} = g(z) = W^*z + c$
linear embedding along the principle components.
NN: $z = f(x) = \sigma(Wx + b)$, $\hat{x} = g(\hat{a}(x)) = \sigma(W^*z + c)$
Latent Space: meaningful DOF, continous and interpolatable
undercomp: compress, features; overcomp: copy components
Not all DOF are important, when we look at the image space (f.e. 256 x 256 x 3), we could as well sample something very random which does not represent an image.
Undercomplete Z: Compressed representation, learns important features of input. Bad for out-of-distribution samples
Overcomplete Z: Hidden units copies input components, but might not extract meaningful structure.

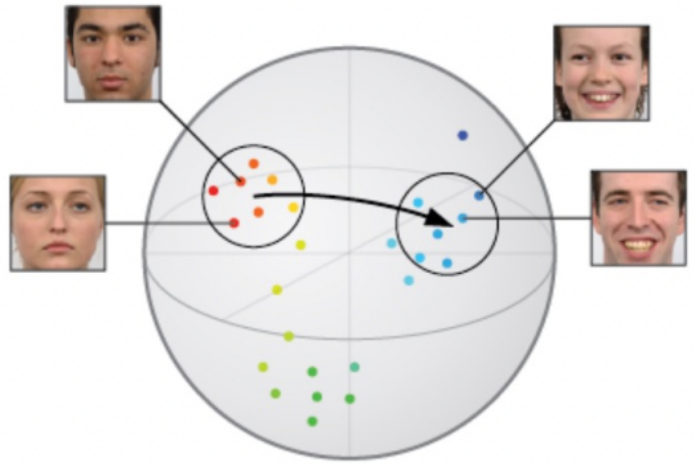


Figure 29: Latent space

Denoise: input+gaussian noise, reconstruct w. overcomp. z
Note: The Latent space is overcomplet in this instance
The model learns components of the images, which it can add to the obstructed image in the end.

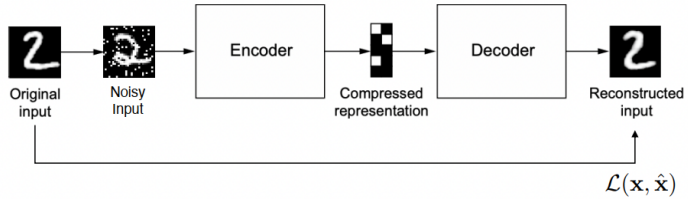


Figure 30: VAE pipeline for denoising

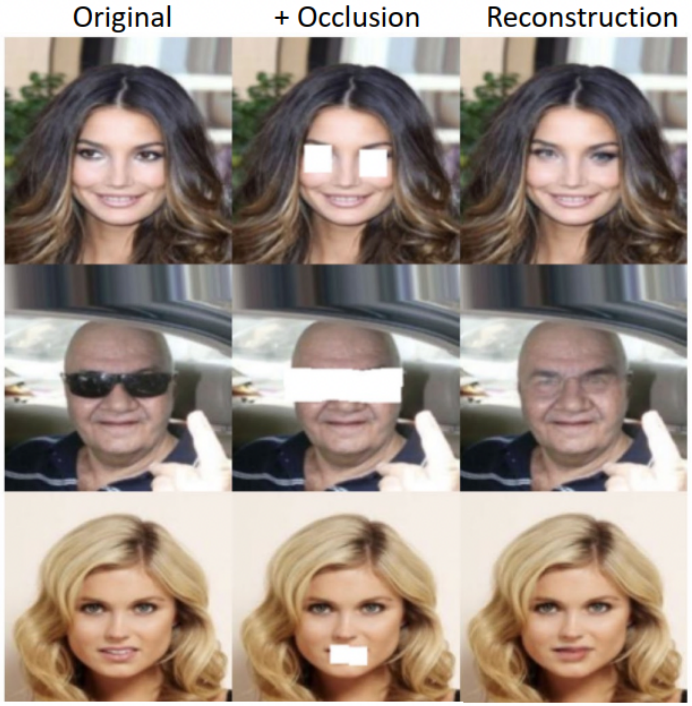


Figure 31: VAE de-noising example

Limit: z not interpolatable; +reconstruction, -new samples
Freshly generated samples from latent space Z do not make much sense, as in these regions, the decoder doesn't know what makes sense. Still, for reconstruction, the latent space is very good.

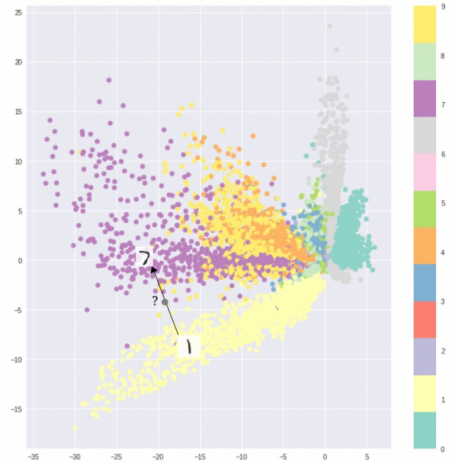


Figure 32: AE MNIST embedding, points in the space

VAE: The latent space Z is approximated by $f(x) \sim \mathcal{N}(\hat{\mu}, \hat{\sigma} I)$

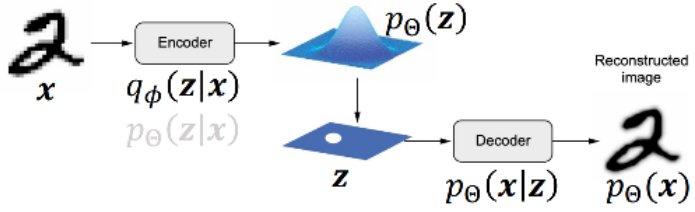


Figure 33: VAE Learned distributions

By design, the latent space is compact and interpolatable. This allows easy sampling and interpolation when generating new samples.

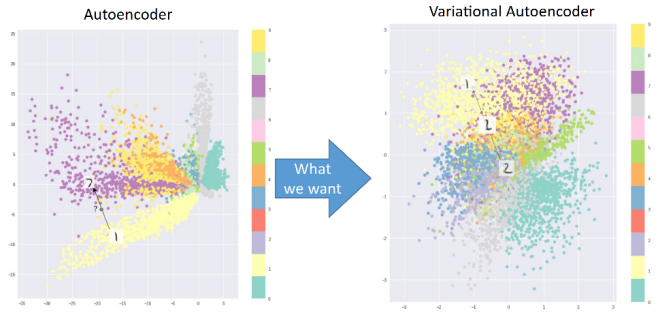


Figure 34: VAE Latent space

Encode: $q_\phi(z|x)$ for $\mu_{(z|x)}, \Sigma_{(z|x)}$, $z|x \sim \mathcal{N}(\mu_{(z|x)}, \Sigma_{(z|x)})$
The Encoder outputs a vector each for μ, σ. This multivariate gaussian can them be sampled to get the latent vector Z depending on x.
Problem: Using only the reconstruction error (MSE), the variance is going to be reduced to zero and the individual classes form clusters around their means. This leads to bad interpolation performance. We want to force the encoder to find features that are as close as possible to each other, while still being distinct.

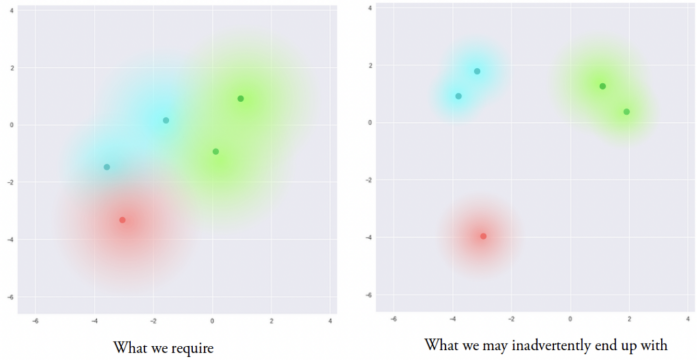


Figure 35: VAE Reconstruction loss only

Decode: $p_\theta(x|z)$ for $\mu_{(\hat{x}|z)}, \Sigma_{(\hat{x}|z)}$, $\hat{x}|z \sim \mathcal{N}(\mu_{(\hat{x}|z)}, \Sigma_{(\hat{x}|z)})$
LH: $p_\theta(x) = \int_z p_\theta(x|z) p_\theta(z) dz$, Post: $p_\theta(z|x) = \frac{p_\theta(X|Z) p_\theta(z)}{p_\theta(x)}$
The decoder, i.e. the conditional $p(x|z)$ is usually a NN, because the output can be really complex, f.e. if we want the generate an image.

KL: $-D_{KL}(q_\phi(z|x)||p_\theta(z)) = \int_x q_\phi(z|x) \log(\frac{p_\theta(z)}{q_\phi(z|x)})$
$= \frac{1}{2} \sum_i^J (1 + \log(\sigma_j^2) - \mu_j - \sigma_j^2)$, if $q \sim \mathcal{N}(\mu, \sigma I), p \sim \mathcal{N}(0, I)$
Properties: asymmetric and non-negative
from exercise: $\int p(z) \log q(z) dz =$
$-\frac{J}{2} \log 2\pi - \frac{1}{2} \sum_i^J \log \sigma_{p,j} - \frac{1}{2} \sum_j^J \frac{\sigma_{p,j}^2 + (\mu_{p,j} - \mu_{q,j})^2}{\sigma_{q,j}^2}$
KL meassures how similar two distributions are. We punish the encoder if the final distribution diverges away from the standart normal. **Note:** Assymetric and non-negative

ELBO: $p_\theta(x) \geq E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p_\theta(z))$
The first part of the loss is the reconstruction loss. It meassures the loss from the decoder prediction.
The second part of the loss is the latent code loss. It enforces that the latent space is a gaussian, e.g. enforcing properties such as compactness and good interpolation
Note: We want to maximize the ELBO
Note: If we have a sequence, the ELBO is the sum of all variational lower bounds
Note: The different classes are still entangled, interpolation works good, but features are not distinguishing unique properties.

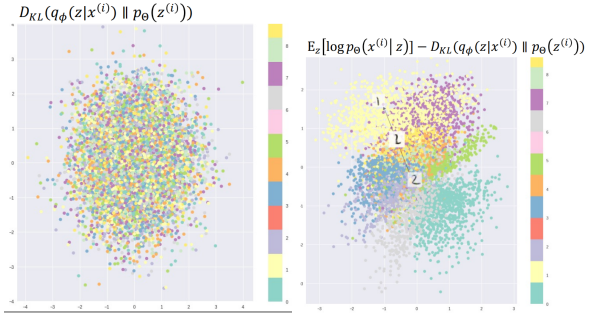


Figure 36: VAE KL-Loss only vs. both losses (entangled)

Re-parameterization: $z = \mu + \sigma\epsilon$, $\epsilon \sim \mathcal{N}(0, 1)$

Goal: Learn features that correspond to distinct factors of variaton e.g., digits and style (or thickness, orientation)
Statistical independence of the of the features can be used
In the picture below, one can see that the AE might achieve detangled features, but it is bad for interpolation. The KL divergence loss craetes some entaglement, though.

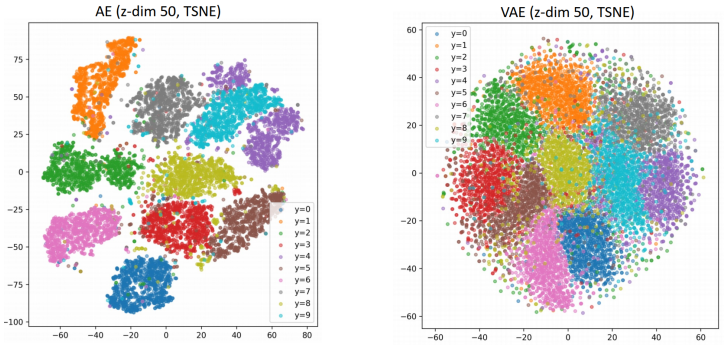


Figure 37: VAE vs. AE

β**-VAE:** $\mathcal{L} = -E_{q_\phi(z|x)}[\log p_\theta(x|z)] + \beta D_{KL}(q_\phi(z|x)||p_\theta(z))$
Goal is to disentangle the different features. The KL-loss tries to create a diagonal multivariate Gaussian, which perfectly seperates the independent features. The reconstruction loss fucks this up, as it fights for the loss as well.
Putting more weight for the KL loss helps solving it in an unsupervised fashion. Alternatively, could condition on features.

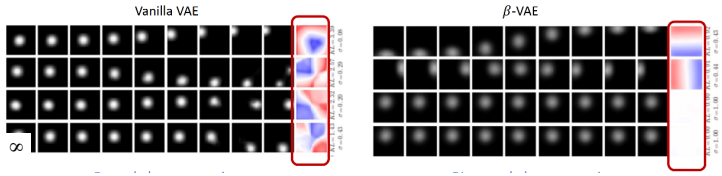


Figure 38: VAE Entangled vs. Disentangled representation

Faces: generated faces are typically blurry

6 Auto-regressive Models

Generativ vs. Discrim.: $P(X|Y) = P(X,Y)($ or $P(X))$

Regressive Property: $x_t = b_0 + b_1 x_{t-1} + b_2 x_{t-2}$

Sequence Model: $p(x) = \prod^N p(x_i|x_1,..,x_{i-1}) = \prod p(x_i|x_{<i})$
+ NLL gives good comparison metric
We are trying to estimate a distribution here.
Note: Sequence models trivially fulfill the regressive property

Believe Net: $\widehat{x}_i = p(x_i = 1|x_{<i}) = Ber(\sigma(\sum_1^{i-1} w_j^i x_j + w_0^i)$
In this fully visible sigmoid belief network, the value of the next element is modelled as a Bernoulli variable.
Note: The weight Matrix $W \in \mathbb{R}^{n \times n}$ is triangular.
Element i at row j contains w_i^j, describing weight of x_i when calculating $\widehat{x}_j$.

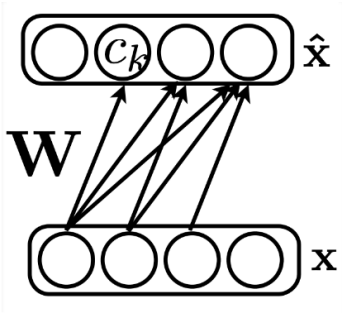


Figure 39: AR Fully Visible Sigmoid Belief Network

NADE: $h_i = \sigma(b + \sum^{i-1} w_j x_j)$ (in $O(H)$, update in $O(1)$)
$\widehat{x}_i = p(x_i = 1|x_{<i}) = \sigma(c_i + V_i h_i)$ (N times $\Rightarrow$ total $O(NH)$)
Train: $\frac{1}{T}\Sigma \log(p(x^t)) = \frac{1}{T}\sum^T \sum^D \log p(x_i^{(t)}|x_{<i}^{(t)})$
The difference to FVSB-Net is that the weights are shared. Therefore, the number of weights is in $\mathcal{O}(n)$.
Note: $b + W_{<i+1}x_{<i+1} = b + W_{<i}x_{<i} + W_{i+1}x_{i+1}$ in $\mathcal{O}(1)$

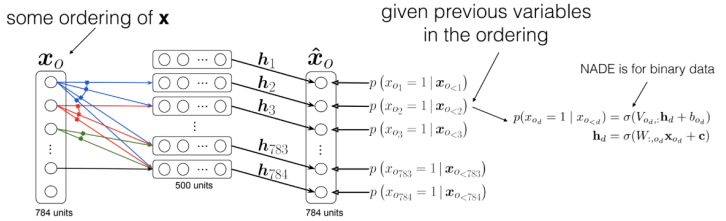


Figure 40: AR NADE Schema

PixelRNN: $p(x) = \prod^{N^2} p(x_i|x_{<i}) = p(x_{i,R}|x_{<i})p(x_{i,G}|x_{<i}, x_{i,R})$
autoreg. from nature: h_t summarises $x_{<t}$; - train/gen is slow
Because of the explicit pixel dependencies from the pixel ordering.
If we use a LSTM for example, $h_{i,j} = f(h_{i-1,j}, h_{i,j-1}, p_{i,j})$, e.g. no parallelization in training, neither in prediction, can be made. It is very slow, but LSTM and RNN can capture long-term dependencies very well.

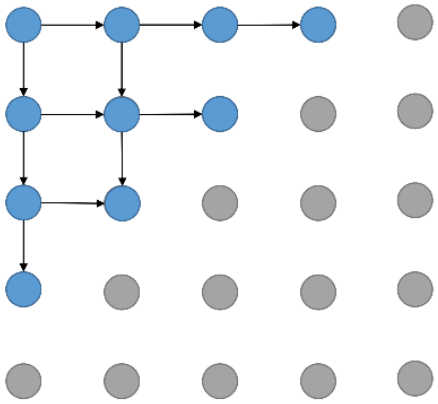


Figure 41: AR PixelRNN dependencies

PixelCNN: Use conv + mask, parallelize training, blind spot
Note: Generation is still sequential
Summary:
Pros are the explicit likelihood p(x) and good evaluation metric through LL of training data.
Cons are slow generation

WaveNet: dilated convolution for large scale temp. dependencies
The issue with sound is that the frequency is at $\sim$ 16'000 samples/second.
To capture speach, convolution must have a very large receptive field.

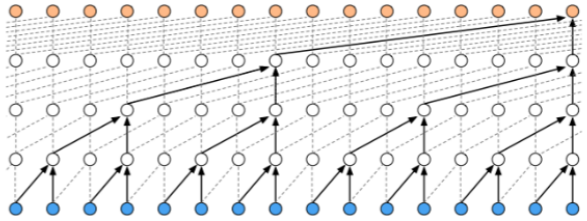


Figure 42: AR WaveNet long term dependencies

Dilated Conv: exponential receptive field size increase

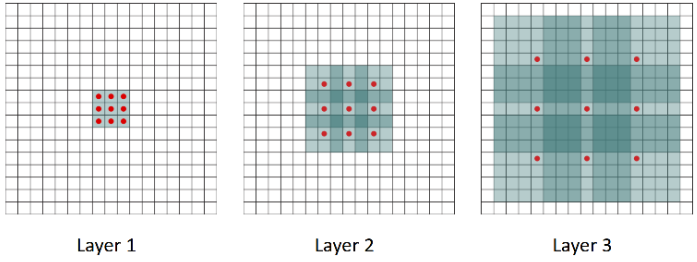


Figure 43: Dilated Conv: Receptive field increase 1

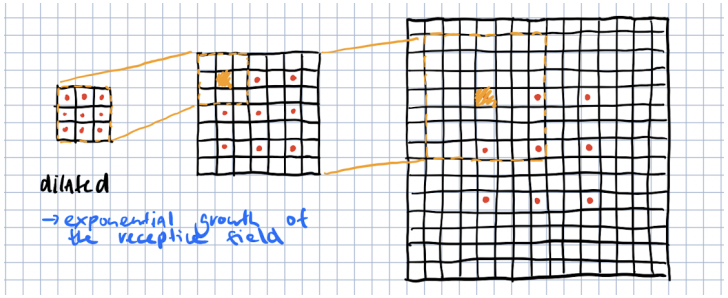


Figure 44: Dilated Conv: Receptive field increase 2

Self-Attention: $K = XW_K, V = XW_V, Q = x_t W_Q (\in \mathbb{R}^D)$,
$X \in \mathbb{R}^{T \times D}, W \in \mathbb{R}^{D \times D}, \alpha = softmx(QK^T/\sqrt{D}), x_{t+1} = \alpha V$
$X = softmax(\frac{(W_Q X)(W_K X)^T}{\sqrt{D}} + M)(W_V X)$
Complexity/path: $O(T^2D)/O(1)$; RNN: $O(nD^2)/O(n)$, Conv: $O(knD^2)/O(\log_k(n))$
RNN: $h_t = g(x_{t-1}, h_{t-1})$, $x_t = f(x_{t-1}, h_t)$. Learns to encode relevant information for future steps and maintains long-term temporal relations.
Complexity per layer: $\mathcal{O}(TD^2)$, with $\mathcal{O}(T)$ sequential operations
CNN: $x_t = f(x_{t-1}, x_{t-2}, .., x_1)$. Learns to aggregate past information for the next step.
Complexity per layer: $\mathcal{O}(kTD^2)$, with $\mathcal{O}(1)$ sequential operations
Self Attention: $x_t = f(x_{t-1}, x_{t-2}, .., x_1)$. Learns to identify relevant information for the next step.
Very expressive, but computationally complex. Attention is a quadratic operation. Complexity per layer: $\mathcal{O}(T^2D)$, with $\mathcal{O}(1)$ sequential operations. Can be restricted in how far we look back, reduces complexity of layer

Figure 45: AR Self-attention

7 Normalizing Flows

+ invertible, +exact LL, +latent space

Goal: What we want is a model that combines the tractable likelihoods from AR models with the latent space of VAE.

Spoiler: NF are invertible and creates a bijective mapping between an observation and latent representation z. It also has a tractable exact log-likelihood.

Variable change 1D: $p_x(x) = p_z(h(x))\,|h(x)|$

2D: $\int\int f(x,y)dxdy = \int\int f(g(u,v),h(u,v))J(u,v)dudv$

The probabilitstic mass must be preserved!

Matrix det. lemma: $det(A+uv^T) = (1+v^TA^{-1}u)det(A)$

Normalizing Flow: $f : \mathbb{R} \to \mathbb{R}$, cont. and invertible

$p_X(x;\theta) = p_Z(f_\theta^{-1}(x))\left|det(\frac{\partial f_\theta^{-1}(x)}{\partial x})\right| = p_z(z)\left|det(\frac{\partial f(z)}{\partial z})\right|^{-1}$

Triangular Jacobian det in $O(d)$, else $O(d^3)$

Example Linear transformation: $z = Ax$

$\Rightarrow p_x(x) = p_z(A^{-1}x)\left|det(A^{-1})\right|$

Note: Not every NN works to model f:

- must be differentiable
- must be invertible
- must preserve dimensionality
- jacobian must be computed efficiently

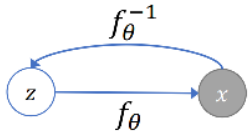


Figure 46: Normalizing Flow mapping

Coupling: $(y_A; y_B)^T = (h(x^A, \beta(x^B)); x^B)^T$ | h:elemnt, β:NN

$(x^A; x^B)^T = (h^{-1}(y^A; \beta(y^B)); y^B)^T$, $J = ((h'; h'f'), (0; 1))$

Note: The Jacobian is triangular. This allows computation of det in $\mathcal{O}(d)$, instead of $\mathcal{O}(d^3)$

Note: β does not have to be invertible

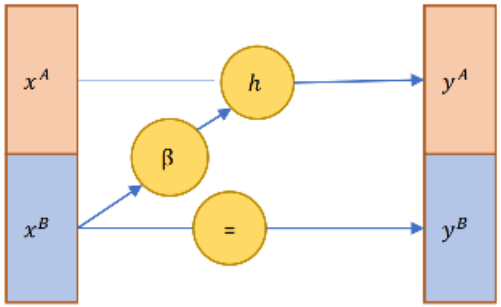


Figure 47: Normalizing Flow coupling layer

Composition: $p_X(x;\theta) = p_Z(f_\theta^{-1}(x))\prod_k \left|det(\frac{\partial f_\theta^{-1}(x)}{\partial x})\right|$

For $x = f(z) = f_k \circ f_{k-1} \circ \circ \circ f_2 \circ f_1(z)$

Train: $logp_x(D) = \sum_x^D (logp_z(f^{-1}(x)) + \sum_k \log\left|det(\frac{\partial f^{-1}(x)}{\partial x})\right|)$

Inference: $z \sim p_z(.)$, $\widehat{x} = f(z)$

The whole point of this is that we want to draw $z \sim \mathcal{N}(0, I)$ and then learn the transformation to create really complex x

Model: $x_i \to squeeze \to n.flow \to split \to z_i$(repeat $L-1$) $\to squeeze \to n.flow \to z_L$, z_i are outputs

Squeeze/Split: reduce spatial dim, pass on half

Flow: actnorm, 1x1 conv, coupling

The flow step is repeated K times and consists of

- actnorm, trainable scale and bias parameter
- 1x1 conv, generalization of a permutation
- affine coupling layer, sometimes with conditional input to β

StyleGAN vs. StyleFlow: Replace mapping Net with normal flow

Imperfect Model: High LL can result in bad samples and vv memorise training data (-LL), high noise samples (+LL)
Question: Is likelihood a good indicator for the quality of samples?
It has been shown that with imperfect models a high likelihood does not imply good sample quality.
Approach: Do not try to model denisity explicitely

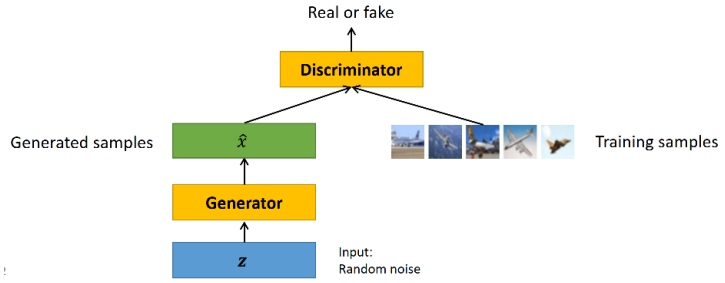


Figure 48: GAN Intuition

Generator: map $z \in \mathbb{R}^Q$ to observ. $x \in \mathbb{R}^D$, $G : \mathbb{R}^Q \to \mathbb{R}^D$
Discriminator: trained on $\widehat{x}$ and x, $D : \mathbb{R}^D \to [0, 1]$
No markov chain necessary
The discriminator can be thought of as learned loss-function.
Loss: $-\frac{1}{2N}(\sum_i^N (y^{(i)}) \log(D(x^{(i)})) + \sum_n^{2N}(1 - y^{(i)}) \log(1 - D(x^{(i)})))$
Train: $G^*, D^* = \arg\min_G \arg\max_D \log(D(x)) + \log(1 - D(\widehat{x}))$
Opt: $V(G, D^*) = \mathbb{E}_{x\sim p_d}(\log(D^*(x))) + \mathbb{E}_{x\sim p_m}(\log(1 - D^*(x)))$
$= -\log(4) + 2D_{JS}(p_d(x)||p_m(x)) = -\log(4)$ if $p_d(x) = p_m(x)$
We can use various NN's for G and D

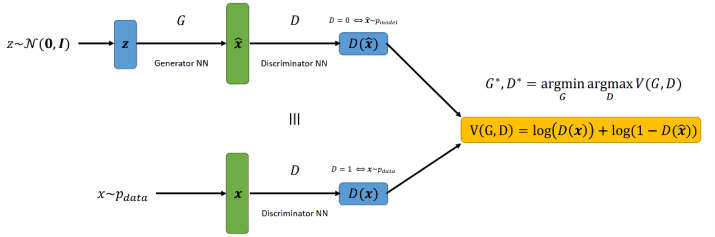


Figure 49: GAN Architecture

$f(x) = a \log x + b \log(1 - x) \in [0, 1]$ has max at $\frac{a}{a+b}$
Update D: for $k : \nabla_{\Theta_D} \frac{1}{N} \sum \log(D(x^{(i)})) + \log(1 - D(G(z^{(i)})))$
Optimum $D^* = \frac{p_{data}(x)}{p_{data}(x) + p_{model}(x)}$
Note: We are using gradient ascent here, because D wants to maximize the loss function
Update G: $\nabla_{\Theta_G} \frac{1}{N} \sum \log(D(G(z^{(i)})))$ (ascent)
Note: We are also using gradient ascent here with a changed objective, instead of using descent on $\log(1 - D(G(z^{(i)})))$, because it has better gradient properties.

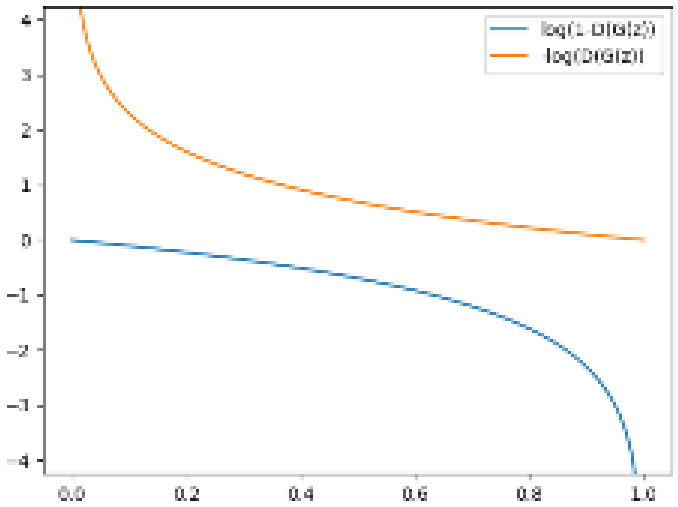


Figure 50: GAN Generator objective

Assumption: capacity, $D \to D^*$, opti. p_{model}
G and D have enough capacity, descriminator reaches D^* in every outer iteration, we optimize p_{model} instead of parameters of the model
Mode Collapse: G finds one mode, D fails to reject
This reduces the set of generated examples, since G has a gold-shitting donkey. If D doesn't figure out to reject it, the next iteration of G again uses it.
Unrolling is updating the parameters of D in the first step, while adjusting the parameters of G at the k'th step. G is looking k steps ahead on how D is reacting, and generates the gradient on each step of the way. Computationaly, we are back-propagating the gradient of G through all k steps, as in LSTM's.

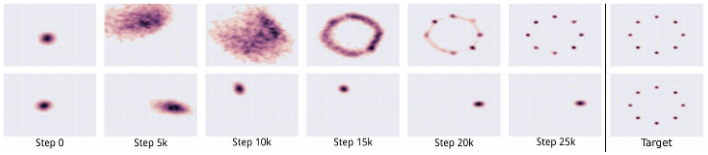


Figure 51: GAN Mode collapse

Oscillation: limited capacity and not D^*
Wasserstein: work to similarity; +no collapse, +stable
Cons: no explicit $p(x)$, nor sample LL, carful balancing, less theory

9 Parametric Body Models

LBS: $t'_i = \sum_k w_{ki} G_k(\theta, J) t_i$
t, t' Rest/ Transformed vertices
w_{ki}: Blend skinning weights (given by designer)
G_k: Rigid bone transformation
θ: Pose
J: Joint locations

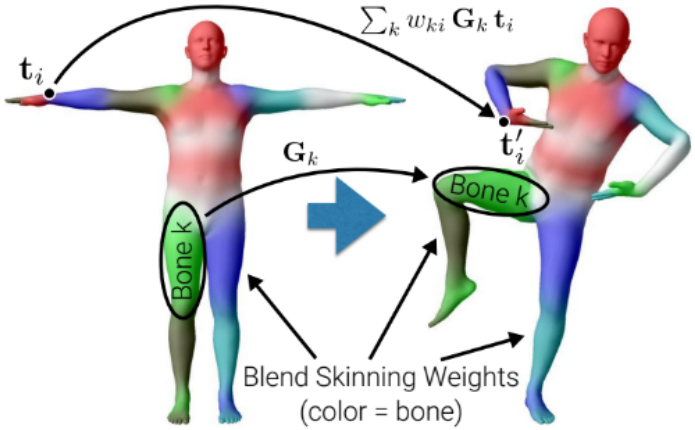


Figure 52: PBM Linear Blend Skinning

MPL: $t'_i = \sum_k w_{ki} G_k(\theta, J(\beta))(t_i + s_i(\beta) + p_i(\theta))$
Differences to the standeart Linear Bend Skinning is that

- Joints J depend on shape beta
- Shape correctives s
- Pose correctives p

Pipeline: Template mesh, joint locations, shape to body shape, add pose correction, LBS

Learned GD: $\Theta^{t+1} = \Theta^t + F(\frac{\partial L}{\partial \Theta}, \Theta^t, x)$

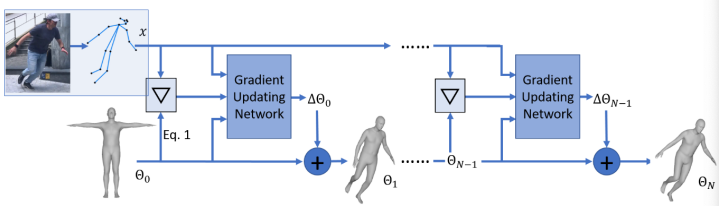


Figure 53: PBM Inference Pipeline

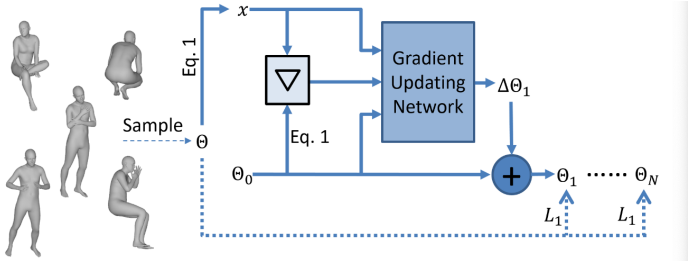


Figure 54: PBM Inference Training

Points to Surfaces
Challenges:

- Self-occlusion
- Lack of depth information
- Articulated motion
- Non-rigid deformation

10 Neural Implicit Representations

Traditional: cam, pointcloud, mesh, tracked mesh

Voxel: 3D grid, limited resolution in $O(n^3)$
If we use a signed distance Field, it has similar properties.

Points: Sensor meassurements, no connectivity/ topology

Mesh: Vertices and surfaces, self-intersections, discont.
A big differentiation can be made between implicit and explicit surfaces. A mesh is explicitely defining vertices and surfaces, which introduces an approximation error, since only a finite amount of elements are used. With an implicit function, we can check wheater a point is on the surface really cheap and quick.

Implicit Repr.: set-level of cont. function, no approx error store SDF values on a regular grid
Neural Impl. Repr: function as NN represents shape; +low memory
This achieves arbitrary topologies and resolutions, the function is also continous. We remove the memory bottleneck like this.
What we want is a function that inputed a location, gives us a distance meassure:

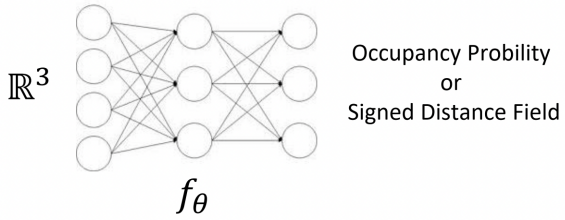


Figure 55: NIR representation

from Mesh: $\mathcal{L}(\theta,\phi) = \sum BCE(f_\theta(p_{ij}, z_i), o_{ij})$, rand. query
p is position, o is occupacy. The MLP learns decision boundary, which concides with inside and outside the mesh here.
from Pointcloud: $\mathcal{L}(\theta) = \sum |f_\theta(x_i)|^2 + \lambda \mathbb{E}_X(\|\nabla_X f_\theta(x)\| - 1)^2$
The Eikonal function (shortest path'ish) with wave velocity 1 is converging to the signed distance function when used with MLP. **Note:** Much harder than from Mesh, since only (noisy) point of the surface to create boundary

Eikonal PDE: $\|\nabla f(x)\| = 1, f(x) = 0, x \in \Omega$, gives SDF Ω
$\mathcal{L}(\theta) = \Sigma_i |f_\theta(x_i)|^2 + \lambda \mathbb{E}_X(\|\nabla_x f_\theta(x)\| - 1)^2$; converges!

Derivating Volume rendering point location + encoded picture, 5 ResNet, Occupancy and Texture head
Occupancy meassures if we are on the surface, behind or in front of the surface.

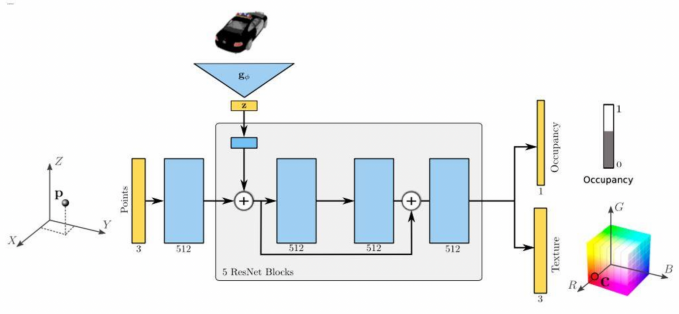


Figure 56: NIR Derivating Volume rendering

Forward: $\forall u$, ray from r_0 through u to root $\hat{p}$ $(\downarrow)$, $u = t_\theta(\hat{p})$
$y_2 = \frac{f(x_1)-f(x_0)}{x_1-x_0}(x_2 - x_1) + f(x_1), x_2 = x_1 - f(x_1)\frac{x_1-x_0}{f(x_1)-f(x_0)}$

Render: shoot rays, rough occupancy est., secant, texture query
We assume to know the camera position, from there we shoot rays though the pixels of the image and query occupancy for equidistant points. Query the color for the intersection point from the texture head.

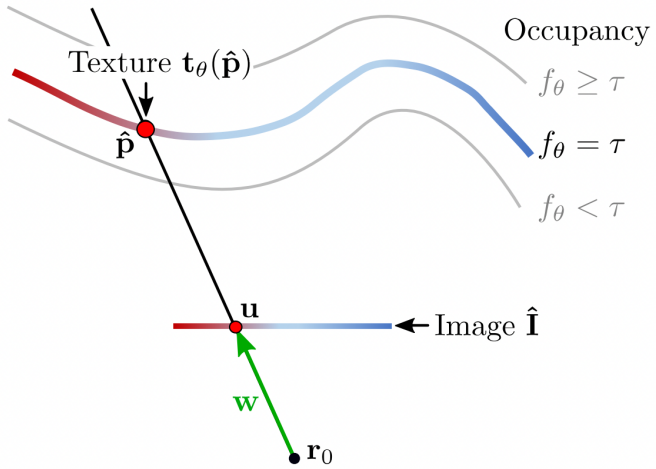


Figure 57: NIR DVR forward pass

Backprob: $\mathcal{L}(\hat{I}, I) = \sum \left\| \hat{I}_u - I_u \right\|$, $\frac{\partial \mathcal{L}}{\partial \theta} = \sum \frac{\partial \mathcal{L}}{\partial \hat{I}_u}\left(\frac{\partial t_\theta(\hat{p})}{\partial \theta} + \frac{\partial t_\theta(\hat{p})}{\partial \hat{p}} \frac{\partial \hat{p}}{\partial \theta}\right)$, $\frac{\partial \hat{p}}{\partial \theta} = -w\left(\frac{\partial f_\theta(\hat{p})}{\partial \hat{p}} w\right)^{-1} \frac{\partial f_\theta(\hat{p})}{\partial \theta}$

NeRF: $F(x, y, z, \theta, \phi) \rightarrow (r, g, b, \sigma)$, whereas F is FCNN w. ReLU
-static only, -slow render, -need many views
Each MLP is overfitted to one object/scene it was trained on, it really is a composition of multiple views.
Input is (x,y,z) and viewing angle (θ, ϕ); output is color and density. Transparency is only dependend on location, color is dependend on both (reflection). The viewing direction is only fed into the network really late, which allows to focus on geometry and only finetune the view dependend color.

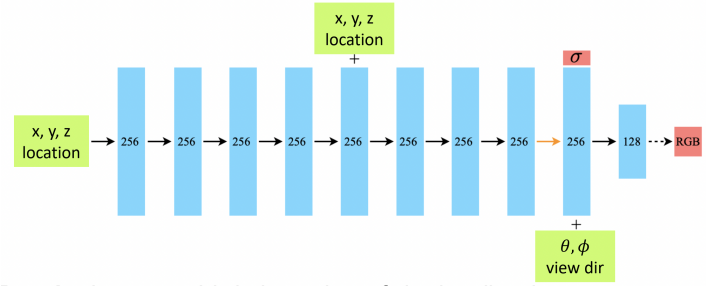


Figure 58: NIR NeRF Architecture

Render: Shoot ray, evaluate all, alpha compose for color
α-composition: $\alpha_i = 1 - e^{-\sigma_i(t_{i+1} - t_i)}$
Transmittance: $T_i = \prod_1^{i-1}(1 - \alpha_j)$, **Color:** $c = \sum T_i \alpha_i c_i$
Alpha is saying if at a certain point is something that emits light (σ), multiplied by a scaling factor. The transmittance is modelling the impact of color down the ray.

Positional encoding: location in fourier space, easier to approximate high frequencies

11 Reinforcement Learning

map states to actions, maximize reward, in uncertain and unknown environment

Return: $G_t = \sum^{\infty} y^k R_{t+k+1}$

Value: $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$
$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = S']]$
The second term is just the weighted sum of rewards given the current state and action. The full thing is just also weighted by the current state and action taken.

Q-Func: $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$, +no trans function
Helps when transition function is not given, as it models the value after a certain action, instead of considering all actions.

Bellman Eq: $v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$

opt: $v_*(s) = \max_a q_*(s, a) = \max_a \sum_{s'} p(s', r|s, a)[r + \gamma v_*(s')]$
What it tells us is that we can decompose the state-value function into immediate reward plus the discounted value of the successor state.

DP: compute optimal policy in perfect model, limited utility
Pros:
- Exact
- Guaranteed to converge in finite time
- Value iterarion more efficient than policy iteration

Cons:
- Need to know transition probability matrix
- Need to iterate over the whole state space
- Memory proportional to state space

Greedy V policy: $\pi'(s) = \arg\max_a (r(s, a) + \gamma V_\pi(p(s, a)))$
... point $V^*(s) = \max_a r(s, a) + \gamma V^*(p(s, a))$, π^* of V^* is π^*
calculate new value function forall state, more efficient
Note: One can show that this greedy policy is always at least as good as any other policy

Policy iteration: Evaluate V with current policy, improve policy

X: +exact, +converges, -trans. prob, -iterate states, -memory

TD learning: $\Delta V(s) = r(s, a) + \gamma V(s') - V(s)$, +less variance
$V(s) \leftarrow V(s) + \alpha \Delta V(s)$, +use only visited states, +effiecient, +no trans. prob, -local min, -biased
Only update value function of visited states.
Random policy would lead to bias in close states. Greedy policy would find rewards quickly, but gets stuck quickly. ϵ-Greedy, use greedy, but with a certain probability choose random.

Pros:
- Less variance than MC
- Sample efficient (no need to update all transitions)
- No need to know transition probability

Cons:
- Biased due to bootstrapping
- Exploration/ Exploitaition?

eps-greedy:. take best action, but random with low prob

SARSA: $\Delta Q(S, A) = R + \gamma Q(S', A') - Q(S, A)$, on-policy
$Q(S, A) \leftarrow Q(S, A) + \alpha \Delta Q(S, A)$
On-policy method, because we are using the current policy to get the rewards from Q

Q-Learning: $\Delta Q(S, A) = R_{t+1} + \gamma \max_a [Q(S', a)] - Q(S, A)$
$Q(S, A) \leftarrow Q(S, A) + \alpha \Delta Q(S, A)$, off-policy
Off-policy, because the policy we use to update the Q function is different from the policy we use to collect the data

Deep Q: $\mathcal{L}(\theta) = (R + \gamma \max_{a'} [Q_\theta(S', a')] - Q_\theta(S, A))^2$
i.i.d assumption, use replay buffer to replay SGD assumes i.i.d: Use replay buffer in training phase and randomly sample transition from the buffer to update. Since Q-learning is off-policy, we are allowed to use old samples
This is a value based method, we try to optimise the value return, thus indirectly exploiting the problem structure to solve the problem. The problem at hand is a regression problem.

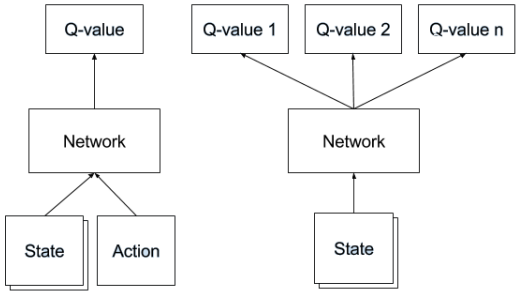


Figure 59: RL Deep-Q network

learn $\pi : S_t \rightarrow A_t$ and $v_\pi : S_t \rightarrow V(S_t)$ as NN

Policy Gradient: $\pi(a_t|s_t) = \mathcal{N}(\mu_t, \sigma_t^2|s_t)$. $p(\tau) = p(s_1) \prod \pi(a_t|s_t) p(s_{t+1}|a_t, s_t)$
Note: The network is predicting μ and σ, which determine the gaussian
We try to make good trajectories more likely, this is a policy method, as we are optimizing the policy directly. Policy methods are used when the action space is really large or continues, we can learn the probabilities for every action directly. It is a classification problem. Compared to Q-learning, it is less sample efficient, because we always need to create new samples and can't just reuse the replay buffer.

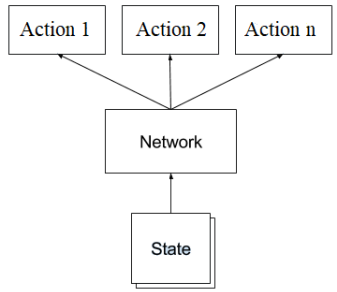


Figure 60: RL Policy Gradient network

Update: $\theta^* = \arg\max J(\theta)$, $\theta = \theta + \nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau)}[(\sum^T \nabla_\theta \log \pi_\theta(a_t^i|s_t^i))(\sum^T y^t r(s_t^i, a_t^i))]$
The first term of the gradient is pointing into the maximum likelihood direction in parameter space, whereas the trajectory reward is scaling this update vector. If the reward is high, we want to make it more likely, if the reward is low, we want to make it less likely.

Reinforce: $\nabla_\theta J(\theta) = \frac{1}{N} \sum_i [(\sum^T \nabla_\theta \log \pi_\theta(a_t^i|s_t^i))(\sum^T y^t r(s_t^i, a b(s_t^i))]$
Use a baseline to reduce the variance of the discounted rewards

Actor-Critic: $\nabla_\theta J(\theta) = \frac{1}{N} \sum_i \sum^T \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)(r(s_t^i, a_t^i) + \gamma V(s_{t+1}^i) - V(s_t^i))$
This is again the temporal difference error