

1 section

1.1 subsection

1.1.1 subsubsection

- some intermediate title

sample text blabla

math : $f : X, x = 43, \forall x. \lambda z$

comment

TODO: todo

code

Table:

c ¹	c ²	c ³
c ⁴	c ⁵	c ⁶

List:

- + pro

- con

Tightcenter:

Text in the center

D.Definition XY here comes the definition

T.Theorem XY here comes a theorem

L.Lemma XY here comes the lemma

Ex.Example XY here comes example

Alg.Algo XY here comes algo

Note: here comes a Note

Important: Here comes something important

Proof: Here comes a proof

Intuition: Here comes some intuition

Raised rule: 

2 Maths

'\t test {test}² okok

test test {asfa | whenever}² okok

test test {test}² okok

test test {asfa | whenever}² okok

3 Introduction

3.1 — Week 1-2: Deep Learning Basics

- MLP
- Fully Connected Networks
- Data, tasks, loss functions
- Backprop
- Activation functions

3.2 — Week 3-4: CNNs, RNNs, co.

- CNNs
- RNNs
- LSTM/GRU/BPTT
- Fully convolutional Networks

3.3 — Week 5-9: Generative Modeling

- Latent variable models
- Implicit Models
- Autoregressive models
- Normalizing Flows/ Invertible Networks

3.4 — Week 10-12: DL 4 CV

- Problems and tasks in human centric computer vision
- DL architectures for CV
- Human body and hand models
- Implicit representations

3.5 — Week 13: Deep RL

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

$\hookrightarrow \text{ILP: } \forall l \quad x^{(l)} = \sigma((w^{(l)})^T x^{(l-1)} + b^{(l)}), f(x; w, b) = x^{(L)}$

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

Softmax: $\text{softmax}(x_i) = \exp(x_i) / \sum_j \exp(x_j)$

Requirements: 1) Output must be positive

2) output must be between $[0, 1]$

3) sum of all outputs must be 1, e.g. $\sum_i^M \text{softmax}(x_i) = 1$

TODO: Linear activation function (Week 2, Page 28)

MLE: Maximise $\log L(\theta) = \log \prod p(x_i|\theta) = \sum \log p(x_i|\theta)$

- 1) Write down probability distribution
- 2) Decompose into per sample probability
- 3) Minimize negative log likelihood

CE loss: $-\frac{1}{N} \sum y_i \log(\sigma(w^T x_i)) + (1 - y_i) \log(1 - \sigma(w^T x_i))$

Note: Cross-entropy loss is a maximum likelihood estimator
We assume the y 's to be bernoulli 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 $|g(x) - f(x)| < \epsilon$

Note: $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ must be non-constant, bounded and continuous

Note: $f \in [0, 1]^m$, e.g. must be in the m-dimensional hypercube

4 Deep Learning

5 Convolutional Neural Network

Goal: Good classification performance.

Trade-off: Specificity vs. invariance

To achieve this, we need to trade off specificity and invariance (created by affine transformations of an object and different lightings).

The Impact (?) is the generalization ability of the model.

Receptive Field: Areas triggering firing of sensory neurons

Note: Areas can be on the Retina, Skin, Tongue, etc.

Note: Usually divided into excitatory and inhibitory regions.

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: Simple cell respond to simple 0-1 input (noisy).

Complex cells, connected to multiple simple cells, form more complex patterns, more resistant to invariance.

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.

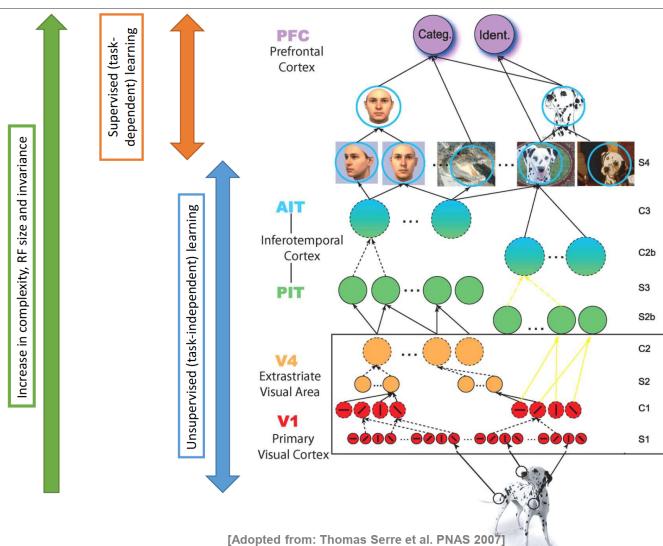


Figure 1: Complexity, unsupervised and supervised learning hierarchy

HMAX: $S: y = \exp(-\frac{1}{2\sigma^2} \sum_{j=1}^{n_{S_k}} (w_j - x_j)^2)$, $C: y = \max_{n_{C_k}} y_j$

x is the input to the cell, w is the weight of the cell. σ tells

us about the size of the receptive fields.

Conv: Modify pixel by some function of surrounding pixels

Note: Any linear, shift-equivariant transform can be written as convolution.

Linear: $T(\alpha u + \beta v) = \alpha T(u) + \beta T(v)$

Invariant: $T(f(u)) = f(T(u))$

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 rotated

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

Lin. Filtering: $I'(i, j) = \sum_{(m,n) \in N(i,j)} K(m, n) I(i+m, j+n)$

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.

1

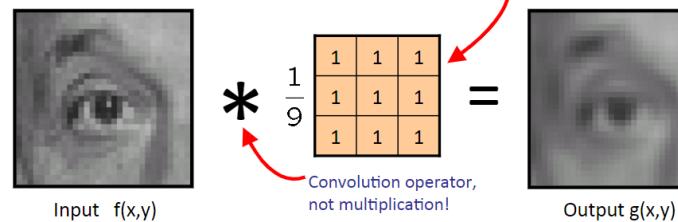


Figure 2: Linear filtering is applied with the convolution operator to compute a new image from the neighbourhood of each pixel of the old image

Correlation: $I'(i, j) = \sum_{m=-k}^k \sum_{n=-k}^k K(m, n) I(i+m, j+n)$

Conv: $I'(i, j) = \sum_{m=-k}^k \sum_{n=-k}^k K(m, n) I(i-m, j-n)$

Alternative representation would be

$I'(i, j) = \sum_{m=-k}^k \sum_{n=-k}^k K(-m, -n) I(i+m, j+n)$

Note: Convolution can be done via matrix multiplication. It acts as a point-spread function, what we want is the weights of this kernel

TODO: Ask how good we must know this, e.g. perform it ourselves?

∇ : Conv. with kernel $[-1, 1]$, $\frac{\delta f}{\delta x} \approx \frac{f(x_{n+1}, y) - f(x_n, y)}{\Delta x}$

Correlation = Convolution: iff $K(i, j) = K(-i, -j)$

Convolutional Layer: $w^T z^{(l-1)} + b$ (+ activation function)

I' dim: $\frac{I_{height} + 2 \cdot \text{padding} - \text{dilation} \cdot (K_{height} - 1) - 1}{\text{stride}} + 1$

Note: If we have 6 filters, the produced "new image" would be of size 28x28x6

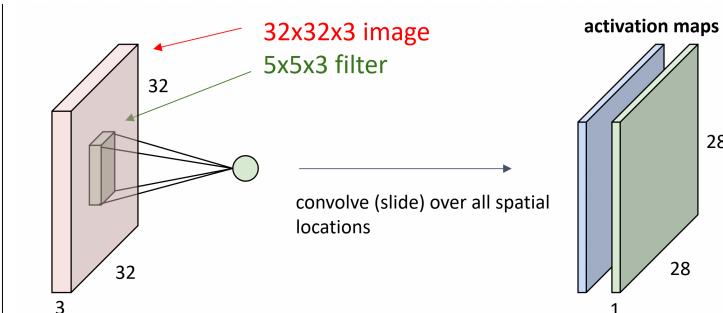


Figure 3: Visualisation applying filters to an image

TODO: Do convolution via matrix on your own!

Weight sharing: Same feature detector on whole image and ↓ weights

Note: This makes the feature detector robust against affine transformations, as it detects the feature in the whole image once trained

Stride: used to reduce size, replaces pooling layers

Dilation: add "holes" to filter, fast increase of rec. field

It is easy to integrate the global context like this

CNN-fwd: $z_{i,j}^{(l)} = w^{(l)} * z^{(l-1)} + b = (\sum_{m,n} w_{m,n}^{(l)} z_{i-m, j-n}^{(l-1)}) + b$

$$\begin{aligned} I'(i', j') &= w * I(i, j) \\ &= \sum_{m,n} w_{m,n} I(i-m, j-n) \end{aligned}$$

Figure 4: Visualisation forward pass

CNN-bwd (z): $\delta_{i,j}^{(l-1)} = \frac{\delta C}{\delta z_{i,j}^{(l-1)}} = \sum_{i',j'} \frac{\delta C}{\delta z_{i',j'}^{(l)}} \frac{\delta z_{i',j'}^{(l)}}{\delta z_{i,j}^{(l-1)}}$

$= \sum_{i',j'} \delta_{i',j'}^{(l)} w_{i'-i, j'-j}^{(l)}$

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

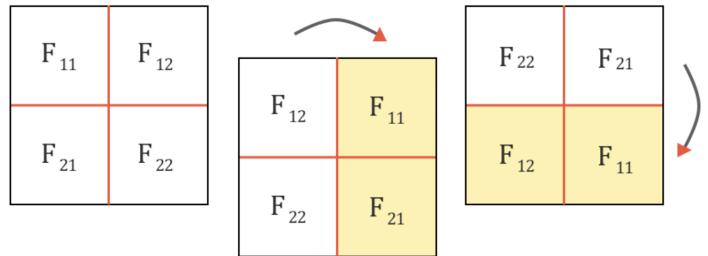


Figure 5: Flipped convolution operator

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

CNN-bwd (w): $\delta_w^{(l)} * ROT_{180}(z^{(l-1)})$

QUESTION: Is this going to be added or substracted to the original image?

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

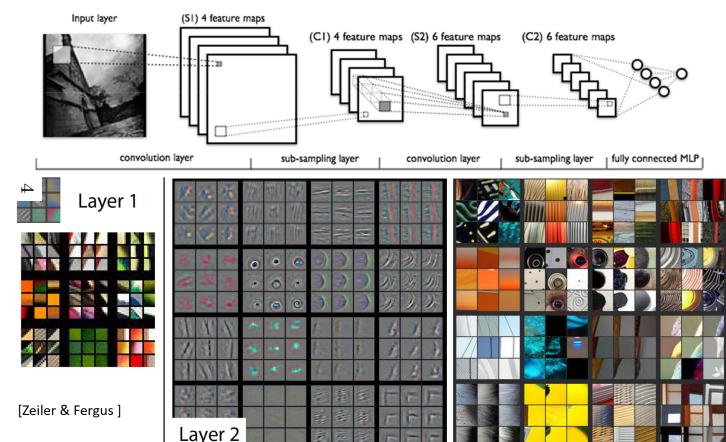


Figure 6: Visualisation of output of CNN layers

Max Pooling: $m \times m$ Filter with m stride, take max. 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.

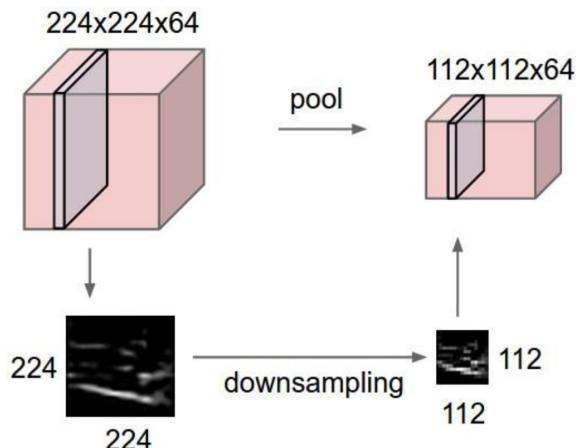


Figure 7: Visualisation of pooling layer

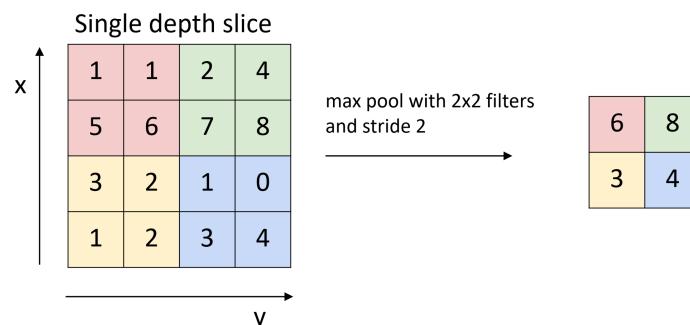


Figure 8: Visualisation of max-pooling layer

fwd: $z^{(l)} = \max\{z_i^{(l-1)}\}$ bwd: $\frac{\delta z^{(l)}}{\delta z_i^{(l-1)}} = 1 \text{ if } i^* = \max\{z_i^{(l-1)}\}$

Only one contributing pixel gets a gradient, all others have gradient zero.

5.1 — Evolution of architectures

Revolution of Depth

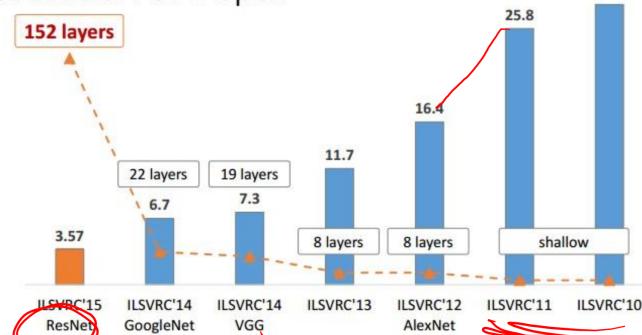


Figure 9: Evolution of NN architecture

5.2 — VGG vs. AlexNet

less kernels/filters (\downarrow params), more layers (\uparrow 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.

5.3 — 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 down-samples 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.

5.4 — 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 belief 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.

• Residual net

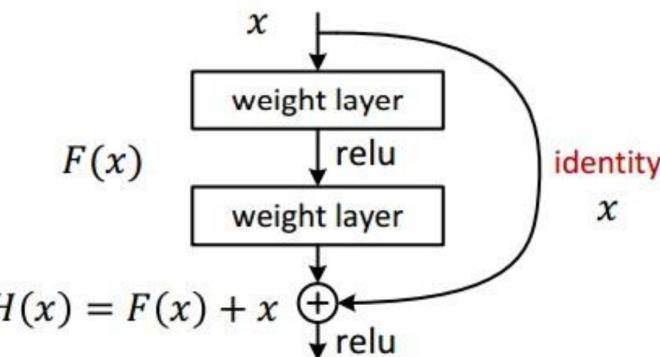


Figure 10: Evolution of NN architecture

6 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.

Goal: Pixel-to-pixel classification By removing the size constraints of the input pictures allows us to classify pixels by using their local neighbourhood.

Drawback: We don't get the information of the whole context. The basic approach is using a standard CNN to learn features of the input image (encoding), which is followed by a decoder who projects the learned features to the higher resolution pixel space.

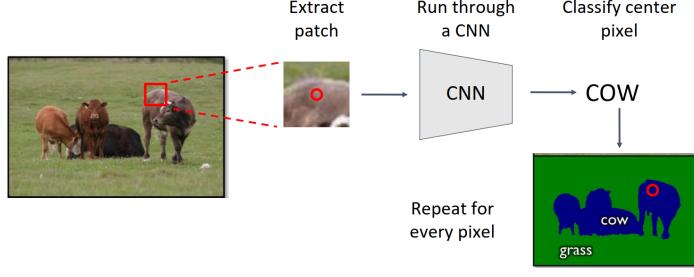


Figure 11: Pixel-wise classification pipeline

Low-res: due to all the pooling and down-sampling
This results in low resolution and fuzzy object boundaries.
Deeper architectures can mitigate this by upsampling again.
The learned features on each downsampled level are copied to the upsampling (decoding) stream, to give the fcn a chance to include features of different levels.

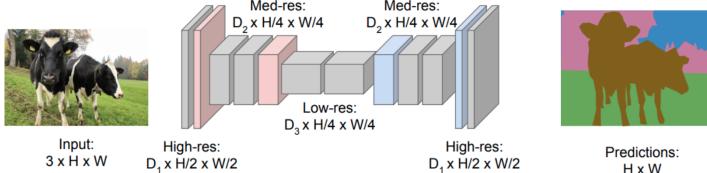
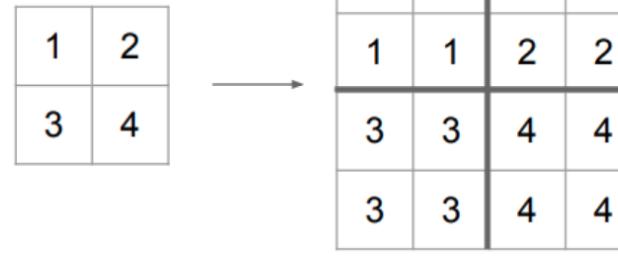


Figure 12: Downsample and then upsample to increase resolution again

Upsampling (NN): copy value for the whole output

Upsampling can be seen as interpolation, increasing the resolution of the signal.

Nearest Neighbor



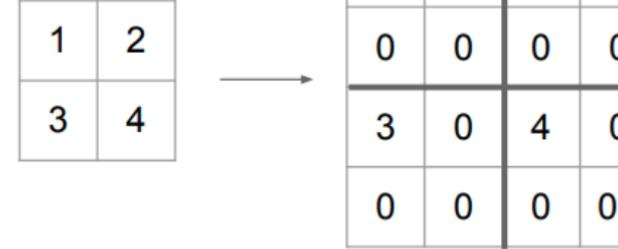
Input: 2 x 2

Output: 4 x 4

Figure 13: Copy input pixel to all output pixels

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

“Bed of Nails”



Input: 2 x 2

Output: 4 x 4

Figure 14: Copy input pixel to one output pixels, rest zero

Max-Unpooling: remember max pooling, BOF to that location

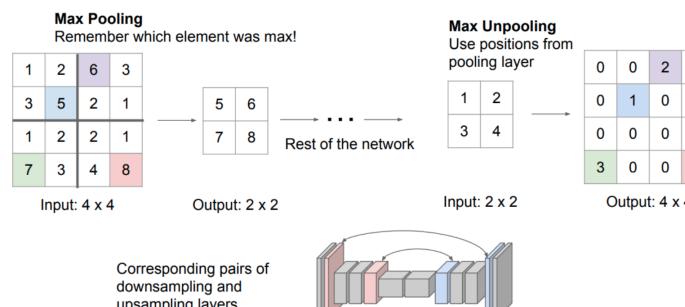


Figure 15: Copy input pixel to location of the max pooling pixel

Transpose Conv: input gives (learnable) weight for filter

3 x 3 transpose convolution, stride 2 pad 1

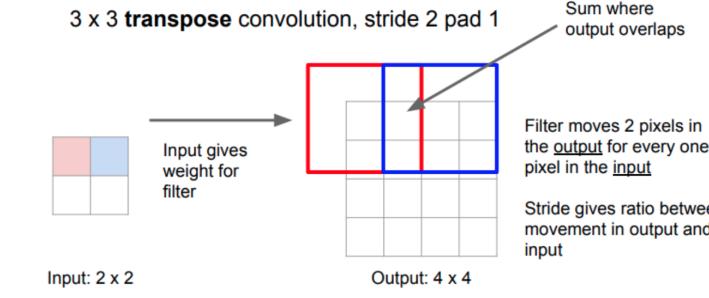


Figure 16: Input pixel gives the weight for a filter

UNet: TransCov + Skip connections

Main idea: combine global and local feature maps by copying corresponding tensors from earlier stages.
Also, the downsampling information is pretty shallow, upsampling performance is improved when including “pre-downsampling” information.

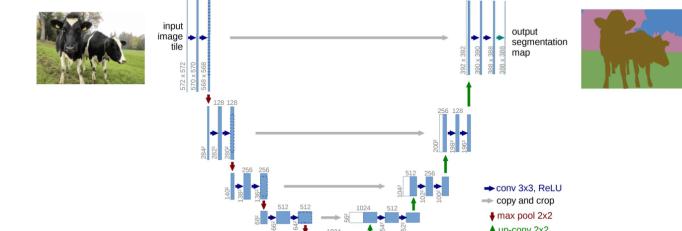


Figure 17: U-Net architecture

7 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 transition given the (implicitly stored) past data.

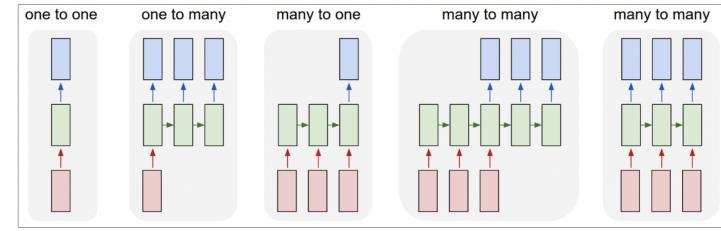


Figure 18: RNN 1. NN, 2. Image captioning, 3. Sentiment analysis, 4. Translation, 5. Video classification

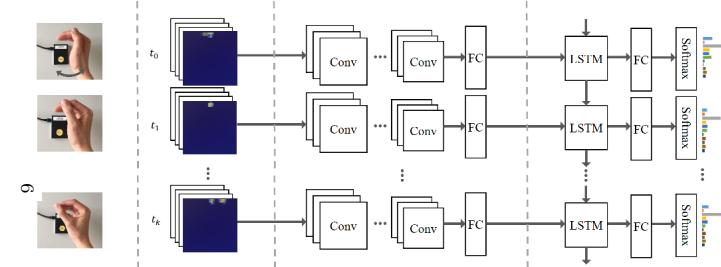


Figure 19: RNN Video classification architecture

Vanilla RNN: $\hat{y} = W_{hy}h^t$, $h^t = \tanh(W_{hh}h^{t-1} + W_{xh}x^t)$

Goal: Learn transition function knowing what's worth keeping. Tanh is important, otherwise we can't forget information.

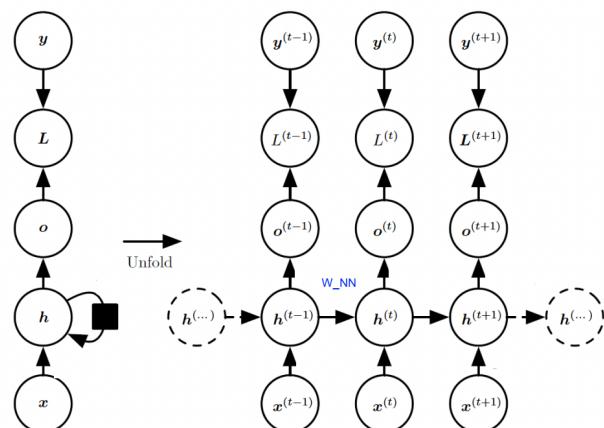


Figure 20: Visualisation of complex RNN

Backprop: $\frac{\delta L}{\delta W} = \sum_{t=1}^S \frac{\delta L^t}{\delta W} = \sum_{t=1}^S \sum_{k=1}^t \frac{\delta L^t}{\delta y^t} \frac{\delta y^t}{\delta h^t} \frac{\delta h^t}{\delta h^k} \frac{\delta h^k}{\delta W}$

TODO: Understand this better

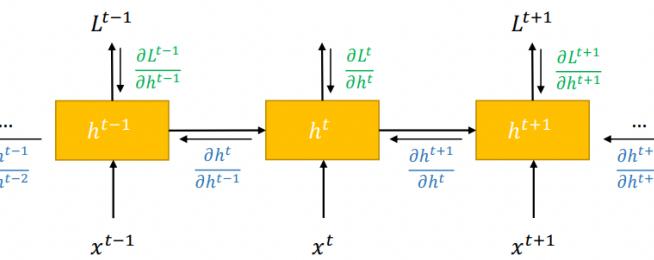


Figure 21: RNN Backprop unrolled

Gradient: $h^t = W^T h^{t-1} = (W^T)^t h^1 = (Q^T \Lambda^t Q) h^1$

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

$$\begin{aligned} \frac{\partial L^t}{\partial W} &= \sum_{k=1}^t \frac{\partial L^t}{\partial y^t} \frac{\partial y^t}{\partial h^t} \frac{\partial h^t}{\partial h^k} \frac{\partial h^k}{\partial W} \\ \frac{\partial h^t}{\partial h^k} &= \prod_{i=k+1}^t \frac{\partial h^i}{\partial h^{i-1}} = \prod_{i=k+1}^t W_{hh}^T \text{diag}[f'(h^{i-1})] \end{aligned}$$

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

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

Figure 22: Proof that vanishing Gradients are a problem

LSTM: $c_t^l = f \odot c_{t-1}^l + i \odot g$, $h_t = o \odot \tanh(c_t^l)$,

$f, i, o, g = [\sigma, \sigma, \sigma, \tanh] \odot W^l [h_{t-1}^l h^{l-1}]^T$

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

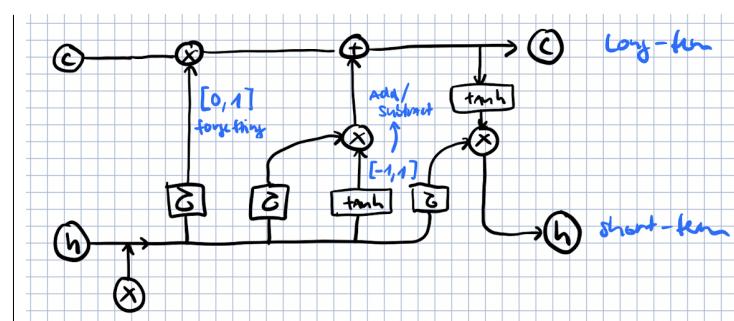


Figure 23: LSTM Visualization

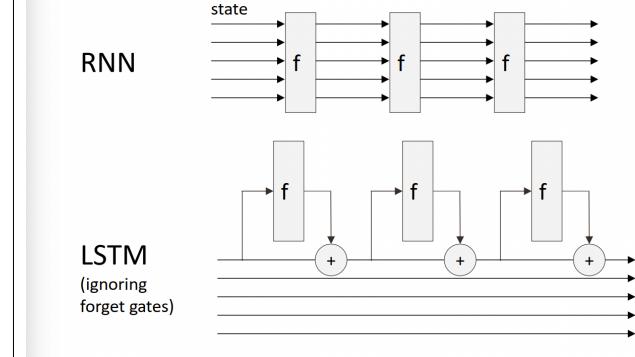


Figure 24: LSTM vs. RNN

8 Variational Auto Encoders

Supervised vs. Unsupervised 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}

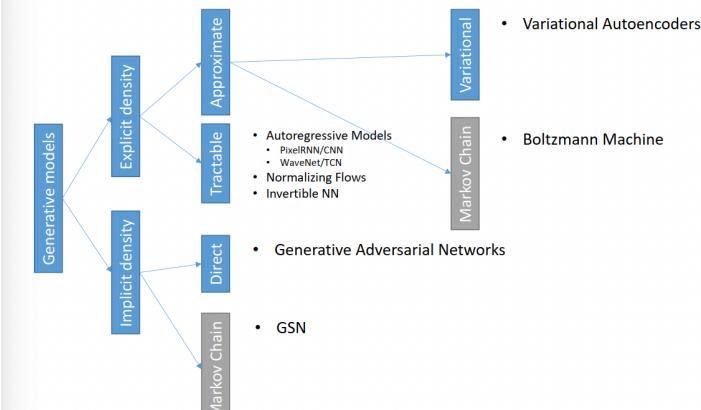


Figure 25: Taxonomy of variational models

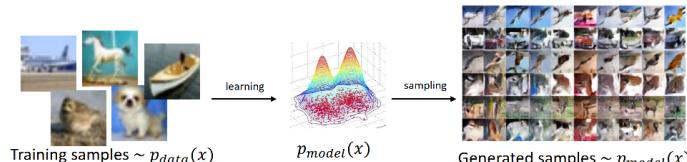


Figure 26: VAE Pipeline

$$\text{AE: optimize } \theta_f, \theta_g = \arg \min \sum_n^N \|x_n - g_\theta(f_\theta(x_n))\|^2$$

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.

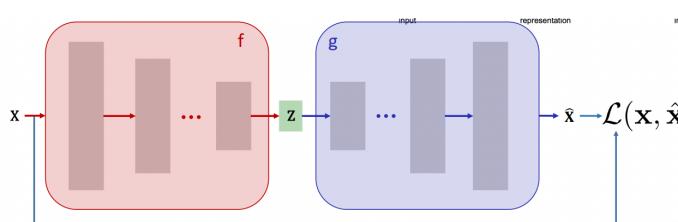


Figure 27: Auto-Encoders Model with loss

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, optimally continuous and interpolatable, under- or overcomplete

Not all DOF are important, when we look at the image space (f.e. $256 \times 256 \times 3$), 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.

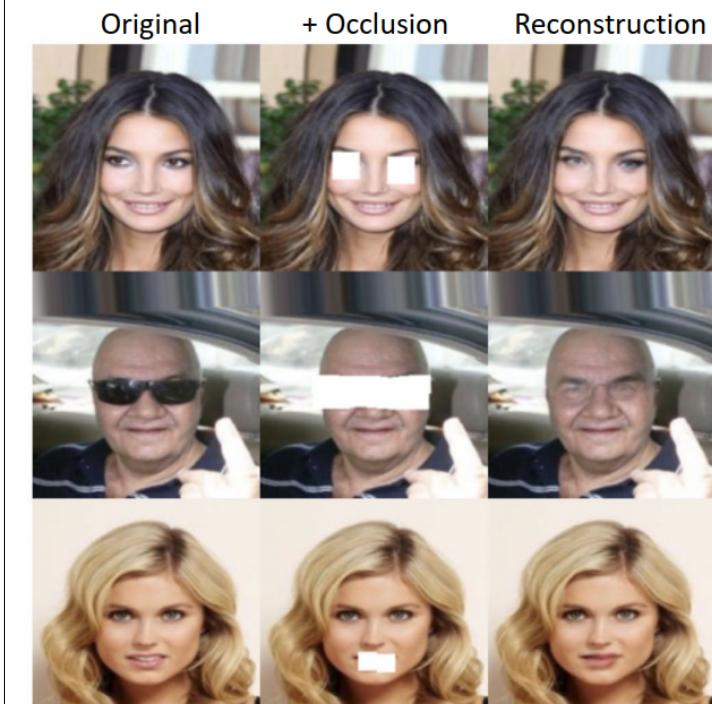


Figure 30: VAE de-noising example

Limit: latent space not interpolatable, unseen 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.

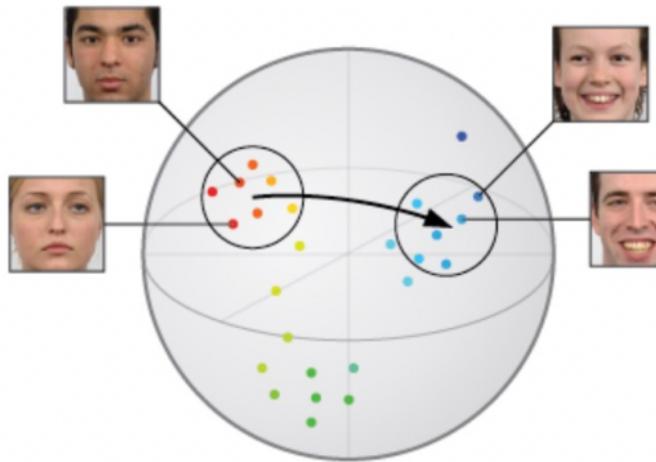


Figure 28: Latent space

Denoising: add gaussian noise to input, reconstruct to original

Note: The Latent space is overcomplet in this instance

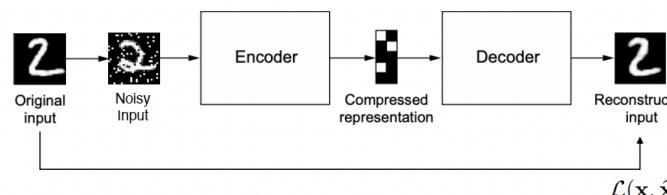


Figure 29: VAE pipeline for denoising

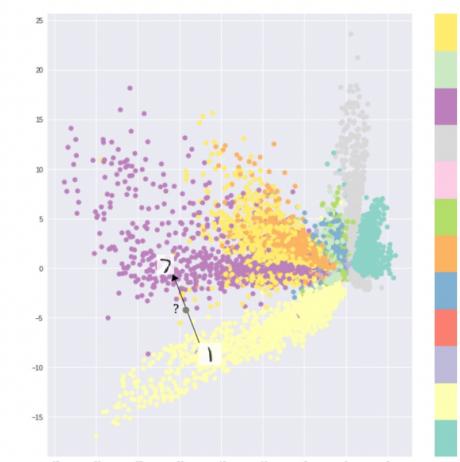


Figure 31: AE MNIST embedding, points in the space

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

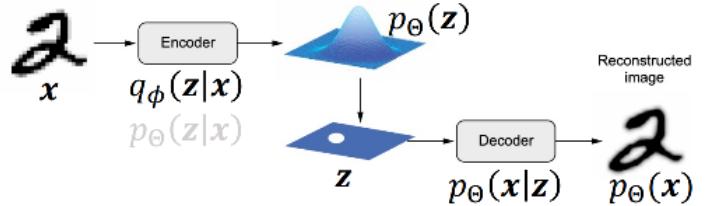


Figure 32: VAE Learned distributions

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

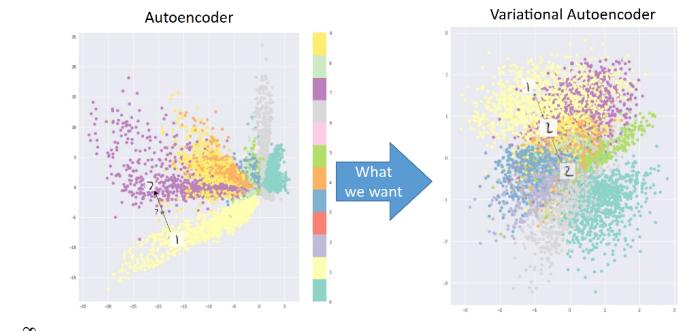


Figure 33: VAE Latent space

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 then 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.

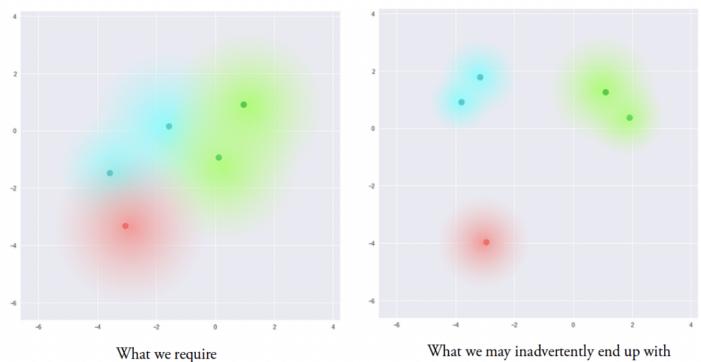


Figure 34: VAE Reconstruction loss only

Decode: $p_\theta(x|z)$ for $\mu_{(\hat{x}|z)}, \Sigma_{(\hat{x}|z)}$, $\hat{x}|z \sim \mathcal{N}(\mu_{(\hat{x}|z)}, \Sigma_{(\hat{x}|z)})$

$$\begin{aligned} \text{KL: } & -D_{KL}(q_\phi(z|x)||p_\theta(z)) = \int_x q_\phi(z|x) \log\left(\frac{p_\theta(z)}{q_\phi(z|x)}\right) \\ & = \frac{1}{2} \sum_i^J (1 + \log(\sigma_j^2) - \mu_j - \sigma_j^2), \text{ if } q \sim \mathcal{N}(\mu, \sigma I), p \sim \mathcal{N}(0, I) \\ \text{from exercise: } & \int p(z) \log q(z) dz = \\ & -\frac{J}{2} \log 2\pi - \frac{1}{2} \sum_i^J \log \sigma_{p,j} - \frac{1}{2} \sum_j \frac{\sigma_{p,j}^2 + (\mu_{p,j} - \mu_{q,j})^2}{\sigma_{q,j}^2} \end{aligned}$$

KL measures how similar two distributions are. We punish the encoder if the final distribution diverges away from the standard normal. **Note:** Assymmetric 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 measures 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.

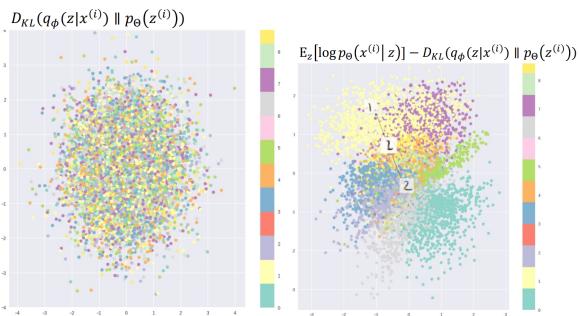


Figure 35: VAE KL-Loss only vs. both losses (entangled)

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

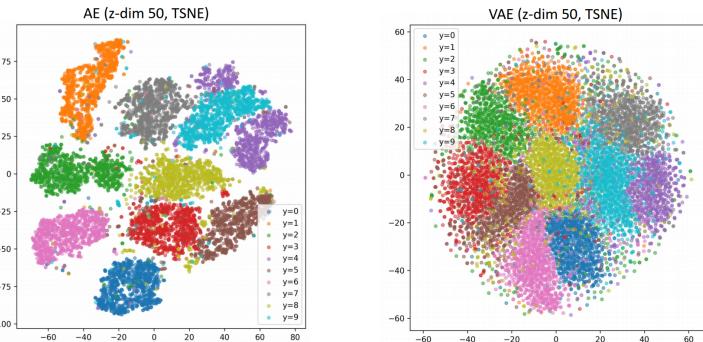


Figure 36: VAE vs. AE

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 separates 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.

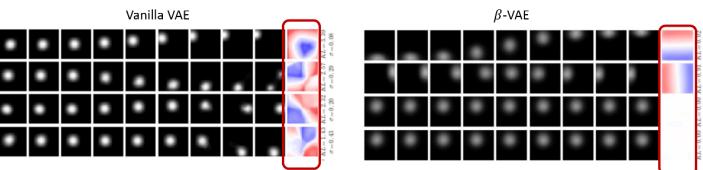


Figure 37: VAE Entangled vs. Disentangled representation

Faces: generated faces are typically blurry

9 Auto-regressive Models

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

Sequence Model: $p(x) = \prod^N p(x_i|x_1, \dots, x_{i-1}) = \prod p(x_i|x_{<i})$

We are trying to estimate a distribution here.

Note: Sequence models trivially fulfill the regressive property

Believe Net: $\hat{x}_i = p(x_i = 1|x_{<i}) = Ber(\sigma(\sum_1^{i-1} w_j^i x_j + w_0^i))$

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_j^i , describing weight of x_i when calculating \hat{x}_j .

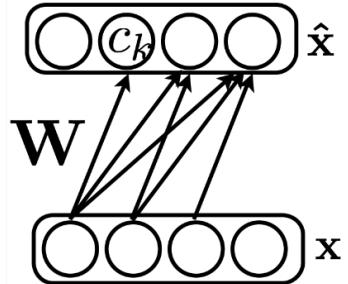


Figure 38: AR Fully Visible Sigmoid Belief Network

• ADE: $h_i = \sigma(b + \sum^{i-1} w_j x_j)$ (in $O(H)$, update in $O(1)$)
 $\hat{x}_i = p(x_i = 1|x_{<i}) = \sigma(c_i + V_i h_i)$ (N times \Rightarrow total $O(NH)$)
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 $\mathcal{O}(1)$

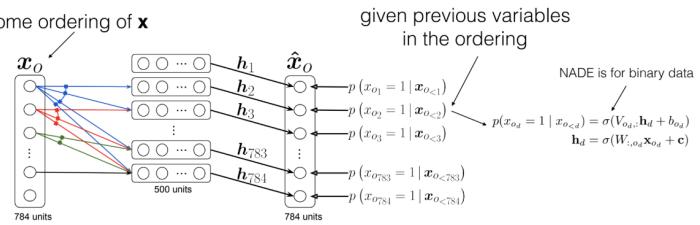


Figure 39: AR NADE Schema

PixelRNN: $p(x) = \prod^{N^2} p(x_i|x_{<i}) = p(x_{i,R}|x_{<i})p(x_{i,G}|x_{<i}, x_{i,R})$
 RNN's are autoreg. from nature: h_t summarises inputs $< t$
 Learning and generation 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.

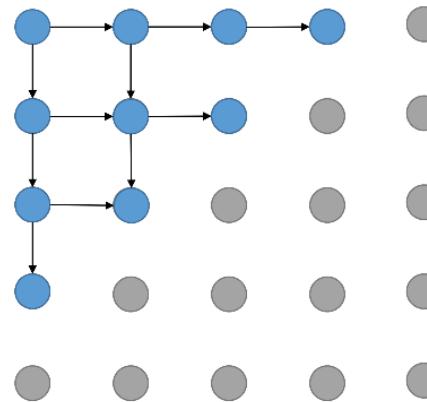


Figure 40: AR PixelRNN dependencies

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 speech, convolution must have a very large receptive field.

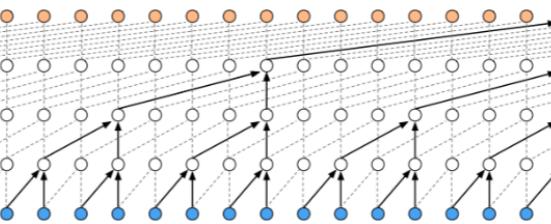


Figure 41: AR WaveNet long term dependencies

Dilated Conv: exponential receptive field size increase

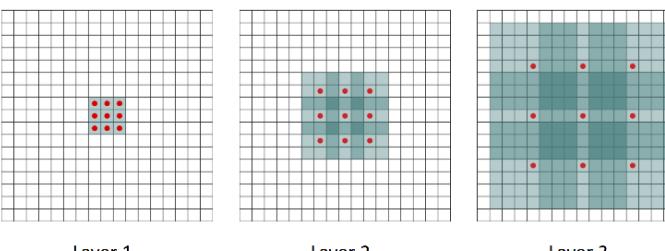


Figure 42: Dilated Conv: Receptive field increase 1

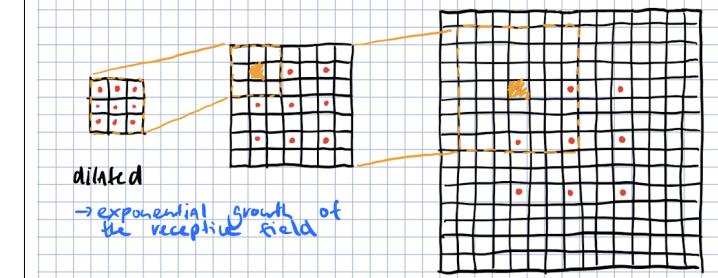


Figure 43: Dilated Conv: Receptive field increase 2

Self-Attention: $K = XW_K, V = XW_V, Q = x_t W_Q (\in \mathbb{R}^D), X \in \mathbb{R}^{T \times D}, W \in \mathbb{R}^{D \times D}, \alpha = softmax(QK^T / \sqrt{D}), x_{t+1} = \alpha V X = softmax(\frac{(W_Q X)(W_K X)^T}{\sqrt{D}} + M)(W_V X)$

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: $x_t = f(x_{t-1}, x_{t-2}, \dots, x_1)$. Learns to aggregate past information for the next step.

Complexity per layer: $\mathcal{O}(kTD^2)$, with $\mathcal{O}(1)$ sequential operations

Self Attention: $x_t = f(x_{t-1}, x_{t-2}, \dots, x_1)$. 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

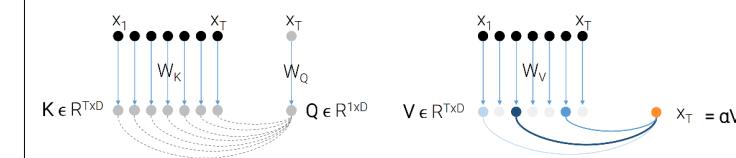


Figure 44: AR Self-attention

TODO: Understand the runtimes better, be able to derive this

10 Normalizing Flows

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: $\int \int f(x, y) dx dy = \int \int f(g(u, v), h(u, v)) J(u, v) du dv$

The probabilistic mass must be preserved!

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

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

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

Example Linear transformation: $z = Ax$

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

Note: Not every NN works to model f :

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

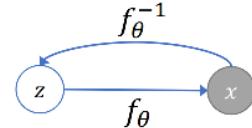


Figure 45: Normalizing Flow mapping

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

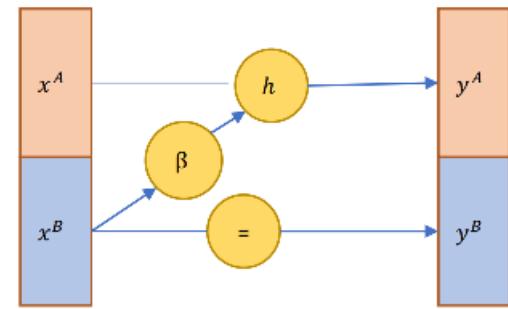


Figure 46: Normalizing Flow coupling layer

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

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

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

Inference: draw $z \sim p_z(\cdot)$, $\hat{x} = f(z)$

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: $x_i \rightarrow \text{squeeze} \rightarrow n.\text{flow} \rightarrow \text{split} \rightarrow z_i$ (repeat $L - 1$)

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

11 Generative Adversarial Networks

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 density explicitly

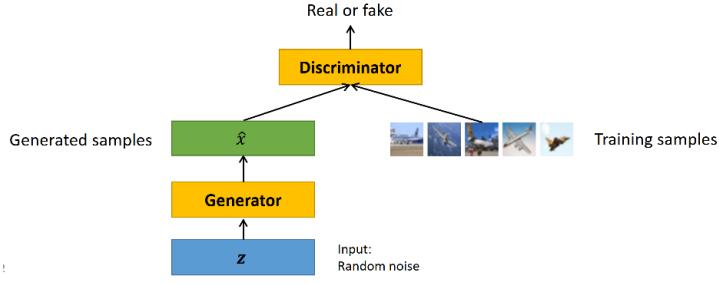


Figure 47: GAN Intuition

Generator: map $z \in \mathbb{R}^Q$ to observ. $x \in \mathbb{R}^D$, $G : \mathbb{R}^Q \rightarrow \mathbb{R}^D$

Discriminator: trained on \hat{x} and x , $D : \mathbb{R}^D \rightarrow [0, 1]$

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: $G^*, D^* = \arg \min_G \arg \max_D \log(D(x)) + \log(1 - D(\hat{x}))$
Intuit: $V(G, D^*) = \mathbb{E}_{x \sim p_d}(\log(D^*(x))) + \mathbb{E}_{x \sim p_m}(\log(1 - D^*(x))) = -\log(4) + 2D_{JS}(p_d(x) || p_m(x)) = -\log(4)$ if $p_d(x) = p_m(x)$

We can use various NN's for G and D

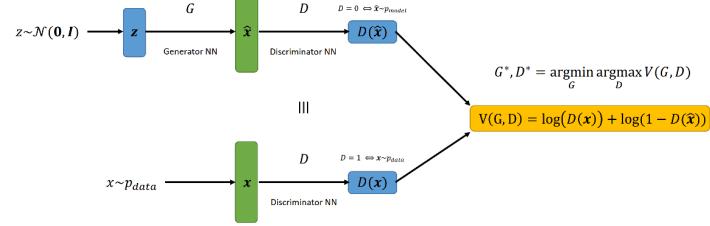


Figure 48: GAN Architecture

$$f(x) = a \log x + b \log(1 - x) \in [0, 1] \text{ has max at } \frac{a}{a+b}$$

Update D: for $k : \nabla_{\Theta_D} \frac{1}{N} \sum \log(D(x^{(i)})) + \log(1 - D(G(z^{(i)})))$

$$\text{Optimum } D^* = \frac{p_{data}(x)}{p_{data}(x) + p_{model}(x)}$$

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.

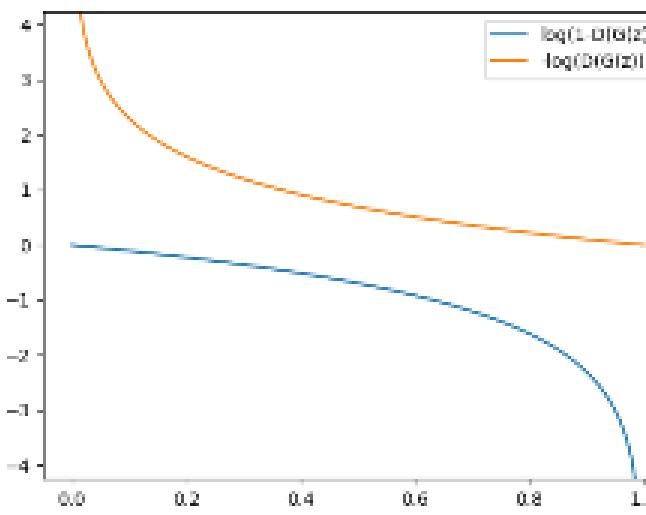


Figure 49: GAN Generator objective

TODO: Why is this true?

Assumption: capacity, $D \rightarrow D^*$, opti. p_{model}

G and D have enough capacity, discriminator reaches D^* in every outer iteration, we optimize p_{model} instead of parameters of the model

Mode Collapse: G figures out one sample that works, 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. Computationally, we are back-propagating the gradient of G through all k steps, as in LSTM's.

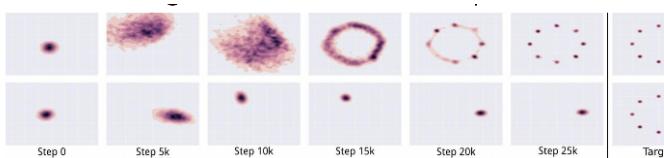


Figure 50: GAN Mode collapse

12 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

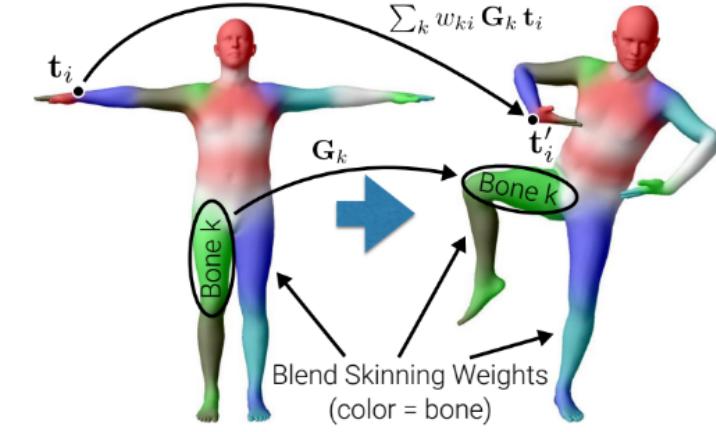


Figure 51: PBM Linear Blend Skinning

MPL: $t'_i = \sum_k w_{ki} G_k(\theta, J(\beta))(t_i + s_i(\beta) + p_i(\theta))$

Differences to the standard Linear Blend 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: $\Theta^{t+1} = \Theta^t + F(\frac{\partial L}{\partial \Theta}, \Theta^t, x)$

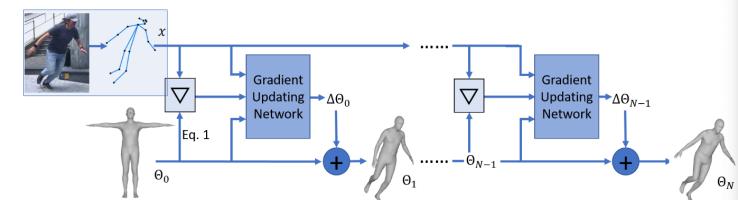


Figure 52: PBM Inference Pipeline

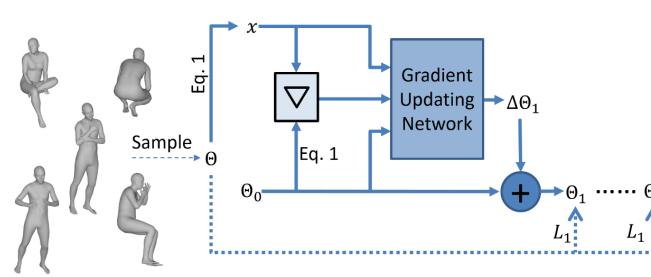


Figure 53: PBM Inference Training

Points to Surfaces

Challenges:

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

13 Neural Implicit Representations

Voxel: 3D grid, limited resolution in $O(n^3)$

If we use a signed distance Field, it has similar properties.

Points: Sensor measurements, no connectivity/ topology

Mesh: Vertices and surfaces, self-intersections

A big differentiation can be made between implicit and explicit surfaces. A mesh is explicitly 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 whether a point is on the surface really cheap and quick.

Implicit Repr.: Surface as set level of cont. function

This achieves arbitrary topologies and resolutions, the function is also continuous. We remove the memory bottleneck like this.

What we want is a function that inputs a location, gives us a distance measure:

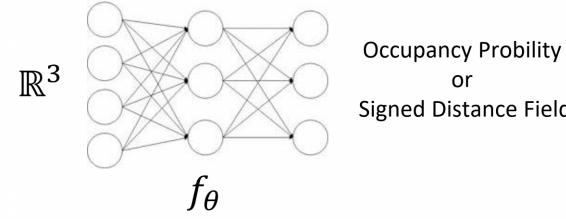


Figure 54: NIR representation

from Mesh: $\mathcal{L}(\theta, \phi) = \sum BCE(f_\theta(p_{ij}, z_i), o_{ij})$

p is position, o is occupancy. 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

TODO: Derivating Volume rendering point location + encoded picture, 5 ResNet, Occupancy and Texture head

Occupancy measures if we are on the surface, behind or in front of the surface.

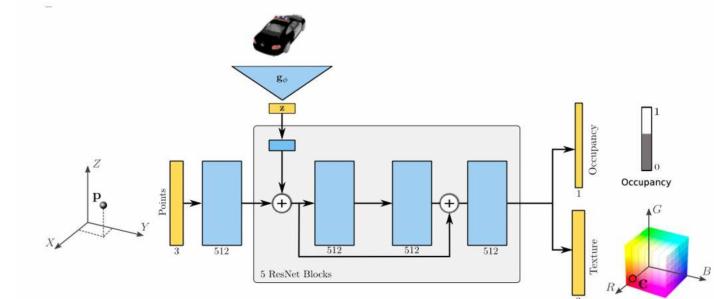


Figure 55: NIR Derivating Volume rendering

Render: shoot rays, rough occupancy est., secant, texture query

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

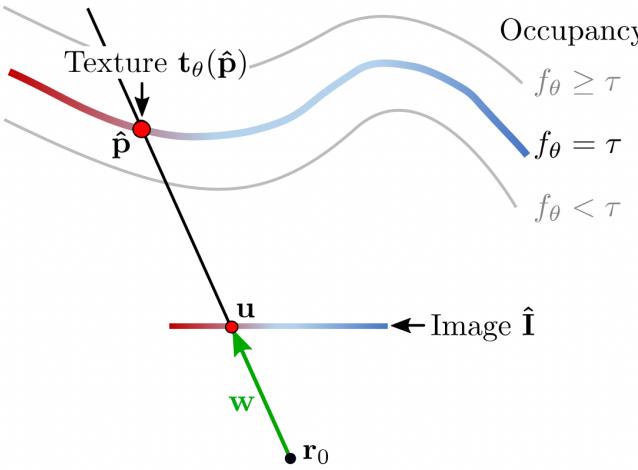


Figure 56: NIR DVR forward pass

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

Secant: $y_2 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1) + f(x_1), x_2 = x_1 - f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)}$

NeRF: $F(x, y, z, \theta, \phi) \rightarrow (r, g, b, \sigma)$, whereas F is FCNN w. ReLU

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.

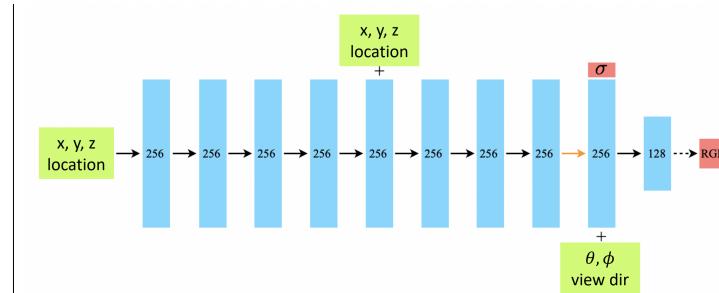


Figure 57: NIR NeRF Architecture

Render: Shoot ray, evaluate all, alpha compose for color

α -composition: $\alpha_i = 1 - e^{-\sigma_i(t_{i+1} - t_i)}$

Transmittance: $T_i = \prod_{j=1}^{i-1} (1 - \alpha_j)$, **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.

14 Reinforcement Learning

Return: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$

Value: $v_{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s] = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s'|s, a)[r + \gamma \mathbb{E}_{\pi}[G_{t+1} | S_{t+1} = s']]$

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: $q_{\pi}(s, a) = \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a]$

Helps when transition function is not given, as it models the value after a certain action, instead of considering all actions.

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

What it tells us is that we can decompose the state-value function into immediate reward plus the discounted value of the successor state.

Greedy value policy: $\pi'(s) = \arg \max(r(s, a) + \gamma V_{\pi}(p(s, a)))$

Note: One can show that this greedy policy is always at least as good as any other policy

DP:

Pros:

- Exact
- Guaranteed to converge in finite time
- Value iteration 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

TD learning: $\Delta V(s) = r(s, a) + \gamma V(s') - V(s), V(s) \leftarrow V(s) + \alpha \Delta V(s)$

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/ Exploitation?

SARSA: $\Delta Q(S, A) = R + \gamma Q(S', A') - Q(S, A), Q(S, A) \leftarrow Q(S, A) + \alpha \Delta Q(S, A)$

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, because the policy we use to update the Q function is different from the policy we use to collect the data

Deep Q: $\mathcal{L}(\theta) = (R + \gamma \max_{a'} [Q_{\theta}(S', a')] - Q_{\theta}(S, A))^2$

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.

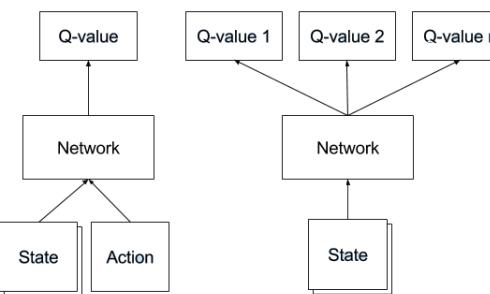


Figure 58: RL Deep-Q network

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 continuous, 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.

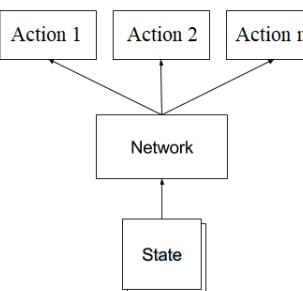


Figure 59: RL Policy Gradient network

Update: $\theta^* = \arg \max J(\theta), \theta = \theta + \nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim p(\tau)}[(\sum_t^T \nabla_{\theta} \log \pi_{\theta}(a_t^i | s_t^i))(\sum_t^T y^t r(s_t^i, a_t^i))]$

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: $\nabla_{\theta} J(\theta) = \frac{1}{N} \sum_i [(\sum_t^T \nabla_{\theta} \log \pi_{\theta}(a_t^i | s_t^i))(\sum_t^T y^t r(s_t^i, a_t^i) - b(s_t^i))]$

Use a baseline to reduce the variance of the discounted rewards

Actor-Critic: $\nabla_{\theta} J(\theta) = \frac{1}{N} \sum_i \sum_t^T \nabla_{\theta} \log \pi_{\theta}(a_t^i | s_t^i)(r(s_t^i, a_t^i) + \gamma V(s_{t+1}^i) - V(s_t^i))$

This is again the temporal difference error