Machine Perception - Summary

FS2024

Machine Perception is the capability of a computer system to interpret their environment in a manner that is similar to the way humans use their senses, to relate to the world around them.

Perceptron: $\hat{y} = f(x, w) = \begin{cases} 1 & \text{if } w^T x + b > 0 \\ 0 & \text{else} \end{cases}$ $(w^T x = \sum_{i=1}^{n} x_i w_i)$

↳ wlog. incorporate bias b into w

Learning algorithm for binary classification:

$k \leftarrow 0$, $w^{(0)} \leftarrow 0$

while $\exists$ a sample $(x^{(i)}, y^{(i)})$ s.t. $\hat{y}^{(i)} \neq y^{(i)}$ do

$w^{[k+1]T} = w^{[k]T} + \eta\,(y^{(i)} - \hat{y}^{(i)})\, x^{(i)}$ where $\hat{y}^{(i)} = \mathbb{1}\{w^{[k]T} x^{(i)} > 0\}$

$k \leftarrow k+1$

end while.

- algorithm converges $\Leftrightarrow$ training set $\mathbb{D}$ is linearly separable.
- no optimality guarantee

Multilayer Perceptron / Neural Network: (extension of Perceptron)
- replace indicator threshold by activation function (non-linear)
- stack mult. Perceptrons into a layer. Layer takes output of previus layer as input.
- $L(\theta)$ loss function is computed with final output

Maximum Likelihood Estimator: Assume we have data $\mathbb{D} = \{x^{(i)}, y^{(i)}\}_N$, and that samples are drawn i.i.d. from unknown distribution p_{data}. Assume the model $p_{model}(y|x,\theta)$ is a family of prob. distr. parameterized by θ.

$$\theta^*_{MLE} = \arg\max_\theta p_{model}(y|X,\theta) = \arg\max_\theta \underbrace{\sum_{i=1}^{N} \log\left(p_{model}(y^{(i)}|x^{(i)},\theta)\right)}_{\text{"log-likelihood"}}$$

→ alternatively: minimize negative log-likelihood (NLL).

Binary Cross-Entropy (as MLE): We can apply MLE to binary classification, by modeling $\hat{y}^{(i)} \sim \text{Bernoulli}(\sigma(\theta^T x^{(i)}))$, where $\sigma(x) = 1/(1+e^{-x}) = e^x/(1+e^x)$ is the Sigmoid activation fct. Thus

$p_{model}(y|X,\theta) = \prod_{i=1}^{N} \left(\frac{1}{1+e^{-\phi}}\right)^{y^{(i)}} \cdot \left(1 - \frac{1}{1+e^{-\phi}}\right)^{1-y^{(i)}}$, denote $\pi^{(i)} = \frac{1}{1+e^{-\phi}}$

$\sigma'(x) = \sigma(x)\cdot(1-\sigma(x))$

can be interpreted as probability, since val. in $[0,1]$

This naturally results in the **cross-entropy loss**

$$\mathcal{L}(\theta) = -\log(p_{model}(y|X,\theta)) = -\frac{1}{N}\sum_{i=1}^{N} y^{(i)} \log(\pi^{(i)}) + (1-y^{(i)}) \log(1-\pi^{(i)})$$

→ Binary cross-entropy: $\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -y^{(i)} \log(\hat{y}^{(i)}) - (1-y^{(i)}) \cdot \log(1-\hat{y}^{(i)}) = \begin{cases} -\log(\hat{y}^{(i)}) & \text{if } y^{(i)} = \\ -\log(1-\hat{y}_i) & \text{if } y^{(i)} = \end{cases}$

Softmax function: can be used for extending the above to multiclass classification. The output of the network is k-dim. and denotes the probability for each class. Softmax ensures that the probabilities add up to 1: $\text{softmax}(x)_i := e^{x_i} / \sum_{j=1}^{k} e^{x_j}$

→ use instead of sigmoid.

MLE Properties:
- If $p_{model}(y|X,\theta)$ is Gaussian $\Rightarrow$ MSE = least squares = $\arg\min_\theta \|x\theta - y\|_2^2$
- If $p_{model}(y|X,\theta)$ is Laplacian $\Rightarrow$ MSE = min. 1-norm = $\arg\min_\theta \|x\theta - y\|_1$
- If prior of θ is Gaussian $\Rightarrow$ MAP = Ridge regression
- If prior of θ is Laplacian $\Rightarrow$ MAP = Lasso regression

Theoretical guarantees:
- Consistency: As sample size $N \to \infty$, MLE converges to true parameter.
- Efficiency: Fast convergence (i.e. unbiased, lowest variance) → among unbiased estimators

Gradient Descent: is used to find local minima of non-convex loss functions $\mathcal{L}(\hat{y}, y)$. (SGD: compute gradient only on one sample) → without closed-form solution

1. Choose learning rate η and tolerance ε.
2. Initialize $\theta^{[0]}$ with small random values.
3. repeat until $\|V\| < \varepsilon$: 3.1. $V = \nabla_\theta \mathcal{L}(\hat{y}, y) = \sum_{i=1}^{N} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$
   3.2. $\theta^{[t+1]} = \theta^{[t]} - \eta \cdot V$

Chain Rule: is used to easily compute derivatives (i.e. gradients) of complex/many concatenated functions. Important for Backpropagation.

For $z = f(g(x)) = f(y)$: $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

Multivariable: $z = f(x(t), y(t))$: $\frac{dz}{dt} = \frac{\partial f}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial f}{\partial y}\cdot\frac{dy}{dt}$

Vectors: $z = f(y) = f(g(x))$, $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\cdot\frac{\partial y}{\partial x} = (\nabla_y z)^T \cdot J_x(y) = \left(\frac{\partial z}{\partial y_1}, \dots, \frac{\partial z}{\partial y_n}\right) \cdot \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix}$$

and $\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j}\cdot\frac{\partial y_j}{\partial x_i}$

Backpropagation: is a way to compute the chain rule, which avoids recomputing the same term several times.

- input to network is passed forward, and gradients backward through layers

Let $\delta^{[l-1]}$ denote the gradient that flows from layer l to $l-1$. For the i-th unit in layer $l-1$, the gradient is

$$\delta_i^{[l-1]} = \frac{\partial \mathcal{L}}{\partial z_i^{[l-1]}} = \sum_{j=1}^{N} \underbrace{\frac{\partial \mathcal{L}}{\partial z_j^{[l]}}}_{=\delta_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial z_i^{[l-1]}}$$

($\mathbb{R}^{1\times N}$, $\mathbb{R}^{N\times M}$)

Thus for the entire layer: $\delta^{[l-1]} = \sum_{j=1}^{N} \frac{\partial \mathcal{L}}{\partial z_j^{[l]}}\frac{\partial z_j^{[l]}}{\partial z^{[l-1]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}}\cdot\frac{\partial z^{[l]}}{\partial z^{[l-1]}}$

Weight update: $\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \sum_{j=1}^{N} \delta_j^{[l]}\cdot\frac{\partial z_j^{[l]}}{\partial W^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}}\cdot\frac{\partial z^{[l]}}{\partial W^{[l]}}$

Block Diagram (MLP): can be used to represent operations in a NN

Activation Function: σ should be non-linear (crucial!). Otherwise the resulting MLP would simply be an affine mapping. → NN becomes useless.

Universal approximation theorem: A MLP with a single hidden layer and continuous non-linear activation function can approximate any continuous function with arbitrary accuracy.

Formally: Let $\sigma: \mathbb{R} \to \mathbb{R}$ non-constant, bounded and cont., $I_m = [0,1]^m$, and $C(I_m)$ cont. fct. on I_m.
Then, any $f \in C(I_m)$, $\forall \varepsilon > 0$, $\exists N, v_i, b_i \in \mathbb{R}$, $w_i \in \mathbb{R}^m$ $(i = 1, \dots, N)$ s.t.

$|g(x) - f(x)| < \varepsilon$ $\forall x \in I_m$, where $g(x) = \sum_{i=1}^{N} v_i\,\sigma(w_i^T x + b_i)$.

In practice, deeper networks work much better. Theoretically, a one-layer NN would have to have infinite width.

Linear Transform: T (e.g. translation, shift operation) is, $\alpha, \beta \in \mathbb{R}$, $u, v \in \mathbb{R}^n$ function f.

- linear: $T(\alpha u + \beta v) = \alpha T(u) + \beta T(v)$
- invariant: $T(f(u)) = T(u)$ wrt. f
- equivariant: $T(f(u)) = f(T(u))$ wrt. f

Any linear, shift-equivariant T can be written as a convolution.

Linear Operation / Correlation: can be represented as follows

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

Where I is the input image, I' the output, K the kernel of dimension $(2k+1) \times (2k+1)$. Note: K does not depend on i,j. This makes I' shift-equivariant. Essentially we move a fixed kernel K over the entire input image I. This process is known as convolution.

I' as above is commonly referred to as Correlation operation.

Convolution Operation: is like the correlation operation, but with kernel K flipped along the x- and y-axes. Formally

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

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

Note that the convolution operation is commutative, thus preferred for implementation.
1-D convolution operation as matrix multiplication: $(I * K) \begin{pmatrix} k_1 & 0 & \cdots & \vdots \\ k_2 & k_1 & & \\ \vdots & k_2 & \ddots & \\ 0 & \vdots & & k_m \end{pmatrix} \cdot \begin{pmatrix} I_1 \\ \vdots \\ I_n \end{pmatrix}$ (Toeplitz-Matrix)

Convolutional layers: in NNs convolve a filter K of size $k \times k \times c$ (with c the input channel size). We can have r such filters, resulting in r so-called activation maps, which are passed on to the next layer. → with additional bias parameter

For an input of dim (c_{in}, H_{in}, W_{in}), we get the following **output size**

$$H_{out} = (H_{in} + 2p_1 - d_1(k_1 - 1) - 1) \cdot s_1^{-1} + 1$$

$$W_{out} = (W_{in} + 2p_2 - d_2(k_2 - 1) - 1) \cdot s_2^{-1} + 1$$

with kernel size $k_1 \times k_2$, padding $p_1 \times p_2$ (i.e. add imaginary border), stride $s_1 \times s_2$ (i.e. skipping input pixels), dilation $d_1 \times d_2$ (i.e. stretch out kernel)

- stride and dilation can increase the receptive field of a network.

probably not relevant

Backpropagation for Convolutional layer: $z_{i,j}^{[l-1]}$ output of $l-1$ layer at pos. (i,j), b bias, $w_{m,n}^{[l]}$ weight of filter at m,n. Then

$$z_{i,j}^{[l]} = W^{[l]} \cdot z^{[l-1]} + b = \sum_{m,n} w^{[l]}(m,n) \cdot z^{[l-1]}(i-m, j-n) + b$$

Derivative of loss function wrt. output of $l-1$ layer

$$\delta^{[l-1]}(i,j) = \frac{\partial \mathcal{L}}{\partial z^{[l-1]}(i,j)} = \sum_{i',j'} \frac{\partial \mathcal{L}}{\partial z^{[l]}(i',j')} \cdot \frac{\partial z^{[l]}(i',j')}{\partial z^{[l-1]}(i,j)} = \sum_{i',j'} \delta^{[l]}(i',j') \cdot w^{[l]}(m,n)$$

where in the last part $i = i' - m$, $j = j' - n$, and thus

$$\delta^{[l-1]}(i,j) = \sum_{i',j'} \delta^{l}(i',j') \cdot w^{[l]}(i'-i, j'-j) = \delta^{[l]} * ROT_{180}(w^{[l]})$$

Thus, we receive the following param. update

$$\frac{\partial \mathcal{L}}{\partial w^{[l]}(m,n)} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial z^{[l]}(i,j)} \cdot \frac{\partial z^{[l]}(i,j)}{\partial w^{[l]}(m,n)} = \sum_{i,j} \delta^{[l]}(i,j)\, z^{[l-1]}(i-m, j-n) = \delta^{[l]} * ROT_{180}(z^{[l-1]}$$

Pooling Layer: reduces the dimension of intermediate output in a NN. (without typically affecting the channel size). This could be done by e.g. summarizing several pixels into one value.
The pooling operation is applied independently to each activation map.
They make data more robust wrt. variance in input data. (Regularization)
Pooling layers locally loose information about precise location (bad is spatial accuracy is crucial).
Examples: Max-pooling, Average-pooling

Max-Pooling: separates the input into $k \times k$ blocks, and summarizes each by taking the maximum value in that block. (like $k \times k$ filter and stride k).

Forward pass: $z^{[l]}(i,j) = \max\{ z^{[l-1]}[(i \cdot s):(i \cdot s + k), (j \cdot s):(j \cdot s + k)] \}$

with k kernel size and s stride. Let (i^*, j^*) be indices of max. value

$$\frac{\partial z^{[l]}(i,j)}{\partial z^{[l-1]}(i',j')} = \begin{cases} 1 & \text{if } (i',j') = (i^*, j^*) \\ 0 & \text{otherwise} \end{cases}, \quad \delta^{[l-1]} = \{\delta^{[l]}\}_{i^*, j^*} \text{ the backward pass.}$$

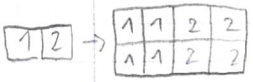
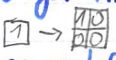
→ the layer has no weights to update. Thus the backward pass is only a propagation of the loss.

ResNet: Is a convolutional architecture introducing residual learning through skip connections. This mitigates vanishing gradients, allowing for much deeper networks. Its improved convergence and higher accuracy have allowed it to set new standards in tasks like image classification.

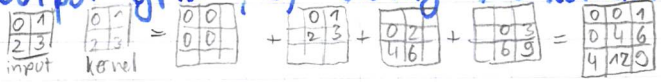
Semantic Segmentation: is the task of assigning a semantic class to each pixel in an image. Naively classifying each pixel by taking a patch around it is highly inefficient. Thus commonly features are downsampled (eg. convolution, pooling) and then upsampled again.

↓ thus input dim. must equal output dim.

Upsampling techniques: are divided into fixed and learnable

Fixed:
- nearest neighbor: duplicate each pixel to get higher resolution 
- bed of nails: pad with zero values 
- Max-unpooling: also pads with zero values, but remembers the location of the maximal values from the max-pooling from downsampling. It puts the non-zero value at this position. 

Learnable:
- transposed convolutions (deconvolutions): Multiply all elements of the kernel by the value of one input pixel. Then add each resulting value to the corresponding pixel position in the output grid, by "sliding" the kernel. Add up overlapping values.

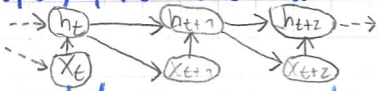
example: 

only!

Dynamical System: describes how a system behaves over time, based on its current state h_t and (sometimes) some input x_t.

Instead of naively modelling temporal dynamics through $h_t = g^{(t)}(x_1, \dots, x_{t-1})$, which leads to many functions to learn, we focus on a simpler approach.

Let $h_t = f(h_{t-1}, x_t; \theta)$, i.e. use same transition function for all time steps. graphically: → learn one model

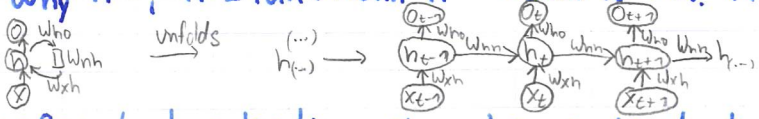


naive: 

- h_t can be interpreted as an internal "memory state"

Vanilla Recurrent Neural Network: is characterized by the vector h_t.

$\hat{y}^t = W_{hy} h^t$, $h^t = \tanh(W_{hh} \cdot h^{t-1} + W_{xh} \cdot x^t + b)$. This unfolds into



→ compared to Sigmoid → faster convergence

- we prefer tanh activation, since it is centered at zero, and it has higher gradients

RNNs are extremely flexible and powerful. They can be used for any combination of One / Many input/output sizes.

Backpropagation in RNNs: Let f denote the activation function (tanh). Write

$h_t = W_{hh} f(h_{t-1}) + W_{xh} x_t + b$, $\hat{y}_t = W_{hy} h_t$, $L_t = \|\hat{y}_t - y_t\|^2$

derivative of loss: $\frac{\partial L}{\partial W} = \sum_{t=1}^{S} \frac{\partial L_t}{\partial W}$ (S: horizon) and $\frac{\partial L_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial^+ h_k}{\partial W}$

because h_t depends on all previous h_k. $\frac{\partial^+ h_k}{\partial W}$ is the immediate derivative. ↳ h_{k-1} is viewed const. wrt. W.

Further note $\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} W_{hh}^T \operatorname{diag}(f'(h_{i-1})) = (W_{hh}^T)^{t-k-1} \prod_{i=k+1}^{t} \operatorname{diag}(f'(h_{i-1}))$

So as our horizon increases, we will regard higher powers of W_{hh}^T. Since $(W_{hh}^T)^{t-k-1} = Q \Lambda^{t-k-1} Q^T$ (singular value decomp), depending on the value of the largest eigenvalue, this term will either explode or vanish.

⇒ Vanishing / Exploding Gradient problem.

(intuitively: Vanilla RNNs are too sensitive to recent distractions)

Gradient Clipping: mitigates exploding gradients, by truncating the gradient update at threshold T. $\theta' = \begin{cases} \theta \leftarrow \theta - \lambda g & \text{if } \|g\|_2 < T \\ \theta \leftarrow \theta - \lambda T \cdot g/\|g\|_2 & \text{else.} \end{cases}$

LSTM - Long Short Term Memory network: are designed to mitigate the exploding and vanishing gradients problem. It is a special kind of RNN, where each cell consists of the following 4 layers:

- forget gate f: scales the previous cell state c^{t-1}. Based on x^t, h_{t-1} it decides (with sigmoid) how much of previous c^{t-1} should be forgotten/kept.
- input gate i: Scales (sigmoid) foreach output of f, how much of x_t, h_{t-1} should be kept for the next cell update.
- output gate o: scales (sigmoid) how much of the cell state c_t should be in the output h_t of the cell.
- gate g: Scales (with tanh (!)) what should be written in the cell state c_t.

In practice, we stack h_{t-1} and x_t and apply one matrix multiplication, and then an activation function, to obtain i_t, f_t, o_t, g_t:

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} \quad \text{where } W = \begin{pmatrix} W_{xi} & W_{hi} \\ W_{xf} & W_{hf} \\ W_{xo} & W_{ho} \\ W_{xg} & W_{hg} \end{pmatrix} \quad \text{and} \quad c_t = f_t \odot c_{t-1} + i_t \odot g_t, \quad h_t = o_t \odot \tanh(c_t)$$

- multilayer case: write $x_t = h_t^{[l-1]}$, $h_{t-1} = h_{t-1}^{[l]}$. (add superscript $[l]$ to all other variables)
- in a Vanilla RNN we would have $h_t^{[l]} = \tanh\left(W^{[l]} \cdot \begin{pmatrix} h_t^{[l-1]} \\ h_{t-1}^{[l]} \end{pmatrix}\right)$

GRU (Gated Recurrent Unit): is a simplified version of LSTM, but maintains similar performance. Let h_{t-1}, x_t be the input to a unit

$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$, $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$, $\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t])$

and finally $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$ the output.

Ch: Training

Regularization: is any modification to a learning algorithm, intended to improve its generalization error, but not its training error. (↳ push fct. in direction of true fct class.)

Parameter Norm Penalties: add penalty term (dependent on θ) to loss fct.

i.e. $\tilde{L} = L + \lambda \cdot \Omega(\theta)$, e.g. $\Omega(\theta) = \|\theta\|_2^2$ or $\Omega(\theta) = \|\theta\|_1$ (↳ prefers sparse solution)

(↳ typically preferred in NN)

Ensemble Methods: uses more than one ML model at the same time for better performance

- train different model types on same data, then aggregate predictions
- train same model type on different data, then aggregate predictions

[Ensemble Methods]

- Bagging: train k models (classifiers) on k bootstraps (one each) (data sampled from original dataset with replacement). Final prediction is combination of k predictions. → models are independent
- Dropout: randomly ignore neurons during training phase. → more robust network
  In some sense it only trains a small percentage of many models that share weights.
  i.e. $r_j^{[l]} \sim \text{Bernoulli}(p)$, $\tilde{y}^{[l]} = r^{[l]} \odot y^{[l]}$, $z^{[l+1]} = \theta^{[l+1]}\tilde{y}^{[l]} + b$, $y^{[l+1]} = f(z^{[l+1]})$

  Dropout testing: (preffered →)
  - take average pred. of forward passes with dropout enabled.
  - Weight Scaling Inference rule: expected total input should be equal in training and testing. i.e. scale $\tilde{\theta} = p\theta$. → works well in practice
    It approximates the true geometric mean of the ensemble.

Data Normalization: Scale input and output to make training/predicting more stable.
use $\mu = \frac{1}{n}\sum_{i=1}^{n} x^{(i)}$, $\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x^{(i)}-\mu)^2}$ → $X_{new} = \frac{X-\mu}{\sigma}$

- important: use same μ, σ across all of training! → collect rolling average during training, for later testing.

Batch Normalization: tries to solve internal covariate shift, i.e. distribution of each layers' input changes during param. update. It scales each mini-batch of intermediate activations: $z_1, \dots, z_n$

$z_i^{norm} = \frac{z_i - \mu}{\sqrt{\varepsilon + \sigma^2}}$, $\tilde{z}_i = \gamma z_i^{norm} + \beta$, ε for numerical stability, γ, β determine mean and Var. at that layer.

- effect on covariate shift is debated.
- BN makes training landscape significantly smoother → better behavior.
- the bias term in a layer becomes redundant with BN.
- weights in deeper layers are more robust to input changes
- slight regularization effect throug addition of inherent noise at each layer.

Data Augmentation: expand dataset by transforming (eg. translation, rotation, flipping, scaling, cutting, noise injection etc.) existing samples.

Pre-Training (Transfer Learning): mitigates problem of small dataset.
First train model on different task, with larger dataset.
Then fine tune trained model on original task.

- init. parameters can have significant effect on regularization.

Pre-existing Architectures: can be incorporated in the model architecture. e.g. with trained weights.

Activation Functions: introduce crucial nonlinearities into NN model.

- Sigmoid: $\sigma(x) = 1/(1+e^{-x})$, $\sigma'(x) = \sigma(x)\cdot(1-\sigma(x))$
- tanh: $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$, $\tanh'(x) = 1 - (\tanh(x))^2$ $\tanh(x) = 2\sigma(2x) - 1$ → scaled sigmoid
- ReLU: $f(x) = \max(0, x)$ → greatly accelerates convergence
  - Dying ReLU problem: no update when activation is negative.
- Leaky ReLU: $f(x) = \alpha x$ for $x < 0$, x for $x \geq 0$.
- Randomized Leaky ReLU: like LReLU, but during training choose $\alpha \in [a, b]$ at random. In testing: $\alpha = (a+b)/2$.

Optimization Algorithms: Optimization and Learning are two very different things. In Learning we optimize an obj. function J, hoping to improve the performance on an unknown measure P.

- Stochastic Gradient Descent (SGD): Gradient descent on unif. randomly chosen one sample. → introduces randomness
  → unbiased, high variance, risk of overshooting, high computing efficiency.
- Mini-batch GD: perform GD on small batch of data.
  → less variance than SGD (more stable convergence), easily parallelizable
- Polyak's Momentum: accelerates gradient stepping, if high curvature, small, but consistent gradients, or noisy gradients. velocity v accumulates successive gradient elements.
  Previous gradients affect direction more, if α is larger rel. to ε.

optimization algorithms: Polyak's Momentum

update rule: $v = \alpha v - \varepsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta), y^{(i)}) \right)$, $\theta \leftarrow \theta + v$

→ large step size, if gradients keep pointing in same direction. → $\frac{\varepsilon \|g\|}{1-\alpha}$ (limit of step size)

→ has been proved to not converge in very simple case.

- Nesterov's Momentum: solves above problem by giving more weight to gradient, and less weight to velocity factor, i.e. write above $L(f(x^{(i)}; \theta + \alpha v), y^{(i)})$
- Adaptive Learning Rate: Adapting learning rate has significant impact on convergence. Thus try to make smaller steps in "steeper" directions.

→ AdaGrad: try to make step size inversely proportional to past gradients magnitude. large/small gradients → rapid/small decrease in learning rate.

$g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$, $r \leftarrow r + g \odot g$, $\theta \leftarrow \theta - \varepsilon / (\delta + \sqrt{r}) \odot g$.

→ accum. of sq. gradient can lead to excessive and premature decrease in learning rate.

→ RMSProp: modified AdaGrad, to solve above problem, and perform better in non-convex settings. Takes exp. weighted moving average i.e. $r \leftarrow \rho \cdot r + (1-\rho) \cdot g \odot g$. Convergence as fast as AdaGrad.

- Adaptive Moment Estimation (Adam): mixes momentum and adaptive learning rate approaches. It makes unbiased estimates of first and second moments $\hat{m}_t$, $\hat{v}_t$.

$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ → $\hat{m}_t = m_t / (1-\beta_1^t)$

$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ → $\hat{v}_t = v_t / (1-\beta_2^t)$

$\Rightarrow \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \cdot \hat{m}_t$.

→ computationally very efficient, little memory usage, robust to hyperparam.

Learning Rate Scheduling:

- step decay: $\eta_{t+1} = \eta_0 \cdot L^{\lfloor t/\tau \rfloor}$
- exponential decay: $\eta_{t+1} = \eta_0 \exp(-kt)$, $k > 0$
- time-based decay: $\eta_{t+1} = \eta_0 / (1+kt)$, $k > 0$.

Ensembles of Deep NN: can be computationally very expensive. But it is known that local minima are connected by simple curves with almost constant training and testing loss. Thus

→ Fast Geometric Ensembling: follow curves of constant loss, to explore local minima. Do this by using cyclic LR for the last 20% of training. Save checkpoints whenever LR is lowest. → Ensemble these models.

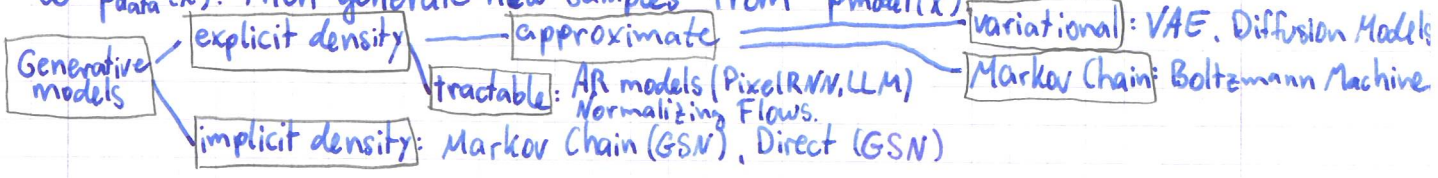
Stochastic Weight Averaging: tries to find wider optima, bcs. they lead to better generalization. For this, take the Deep NN, in the last 20% of training, initialize w_{SWA} from pretrained model, and adopt a cyclic LR for the rest of training.

- every time we reach lowest LR, update running average of w_{SWA}: $w_{SWA} \leftarrow (w_{SWA} \cdot n_{models} + w) / (n_{models} + 1)$
- use w_{SWA} for inference (re-init. batch norm. for w_{SWA}).

→ huge performance increase, normal inference cost, running average is cheap.

(taxonomy)

Generative modelling: We want to learn $p_{model}(x)$ that is similar to $p_{data}(x)$. Then generate new samples from $p_{model}(x)$



Autoencoders: try to find degrees of freedom for efficient data representation. Describe high-dimensional data with low-dimensional representation.

- encoder f: projects original input space X into latent space Z.
- decoder g: maps samples from Z back to X.
- g∘f approximates the identity function on X.
- minimize the objective: $\hat{\theta}_g, \hat{\theta}_f = \underset{\theta_g, \theta_f}{\operatorname{argmin}} \sum_{n=1}^{N} \|x_n - g(f(x_n))\|^2$
- f, g linear => PCA.

Autoencoder - Dimensionality of hidden layer: We call a layer:
- undercomplete: $\dim(z) < \dim(x)$ (→ compression)
  → works very well for training samples. Bad for out-of-distribution samples.
- overcomplete: $\dim(z) > \dim(x)$ (→ no compression)
  → might not extract meaningful features.
  → helpful for denoising and inpainting!
  - Denoising Autoencoders: learned representation should be robust to noise. Randomly set parts of input to 0, and add Gaussian noise. Reconstruction $\hat{x}$ computed from corrupted input $\tilde{x}$, but Loss compares $\hat{x}$ with original input x.
  - Inpainting Autoencoders: learn like above, but from images with missing parts.

⇒ AE is good at reconstruction.

Autoencoder Limitations: For generating new samples, randomly sample from latent space. Gaps in latent space make AE pretty bad at this.
→ solution: VAE

Kullback-Leibler (KL) Divergence: measures how different the distribution p is from a reference distribution q.

$$D_{KL}(p\|q) = \int p(x) \cdot \log\left(\frac{p(x)}{q(x)}\right) dx$$

- not symmetric: $D_{KL}(p\|q) \neq D_{KL}(q\|p)$ (in general)
- non-negative: $D_{KL}(p\|q) \geq 0$ → proof: $-D_{KL}$ and use Jensen: $E[\phi(x)] \leq \phi(E[x])$ (ϕ concave (e.g. log))

Variational Autoencoders: can easily generate new samples, because their latent space is designed to be continuous.
→ Encoder outputs vectors μ and σ (each of dim z). Then z (decoder input) is sampled from $\mathcal{N}(\mu, \sigma)$ on latent space.
→ to ensure that all learned μ are close to each other (i.e. not have distant clusters), and σ is small, introduce D_{KL} into loss fct. We minimize the divergence between $\mathcal{N}(\mu, \sigma^2)$ (for each sample!) and the standard multivariate Normal distr. $\mathcal{N}(0, I)$.
⇒ Maximize the ELBO (with prior $\mathcal{N}(0,I)$, i.e. $P(z)$)

Evidence Lower Bound (ELBO): Generally, we want to maximize data likelihood $P_\theta(x) = \int_z P_\theta(x|z)\, P_\theta(z)\, dz$ ($P_\theta(x|z)$ is decoder). But the integral is intractable. Thus also the posterior $P_\theta(z|x) = P_\theta(x|z) \cdot P_\theta(z) / P_\theta(x)$ is intractable
→ Solution: approximate posterior with an encoder network $q_\phi(z|x)$.

Then maximize → $\log(P_\theta(x)) = \mathbb{E}_{z\sim q_\phi(\cdot|x)}[\log(P_\theta(x))] = E_{z\sim q_\theta(\cdot|x)}\left[\log\left(\frac{P_\theta(x|z)\cdot P(z)}{P_\theta(z|x)} \cdot \frac{q_\phi(z|x)}{q_\phi(z|x)}\right)\right]$

(loss is minimized! ⇒ −ELBO)

$= E_{z\sim q_\phi(\cdot|x)}[\log(P_\theta(x|z))] - D_{KL}(q_\phi(z|x)\|P(z)) + \cancel{D_{KL}(q_\phi(z|x)\|P_\theta(z|x))}$

- reconstruction loss term → Decoder network (use differentiable sampling) → form clusters
- has closed form solution → spread evenly around origin.
- intractable, but ≥ 0

Reparametrization Trick: Let $z \sim \mathcal{N}(\mu, \sigma^2)$. Then we can write $z = f(x, \varepsilon, \theta) = \mu + \sigma\cdot\varepsilon$, $\varepsilon \sim \mathcal{N}(0,1)$ (see ε as a random noise term)
→ This makes the operation of getting z from a random sampling operation, to a deterministic operation → we can backpropagate. (→ i.e. take derivative w.r.t. μ, σ.)

Generating new data: for VAE, sample $z \sim \mathcal{N}(0, I)$ and apply decoder.
→ But obtained representations are still entangled, i.e. there's now way to explicitly sample e.g. an image of a 1 or of a 9.
→ We want to disentangle the latent space. Each dimension should correspond to some feature in the data (e.g. digit, style, thickness, orientation, ...)

Solutions:
- semi-supervised learning: make model learn the corresp. label.
- β-VAE (unsupervised): Give more weight to D_{KL} in loss, to more strongly enforce independence between dimensions in the latent space (i.e. diagonal cov-matrix σ).

comes from constrained optimization $< \delta$ → new loss: $E_{z\sim q_\phi(\cdot|x)}[\log(P_\theta(x|z))] - \boxed{\beta} \cdot D_{KL}(q_\phi(z|x)\|P(z))$

VAE-Limitations: Tend to generate blurry images, probably due to injected noise, and weak model capacity.

Autoregressive Property: Inputs are taken as observations at previous time steps ("lag variables"). It uses data from the same input variable at prev. steps.

Sequence Model: is the modelling of sequential data. They are considered generative models, because given $x_1,\dots,x_k$ predict x_{k+1}, then given $x_2,\dots,x_{k+1}$ predict x_{k+2} and so on.

Parameterizing sequence models: imagine black-white image of n pixels. → 2^n states.
We want to parameterize $p(x_1,\dots,x_n)$ (which is easy to learn)
- tabular approach: factorize joint distribution $p(x) = \prod_i p(x_i \mid x_1,\dots,x_{i-1})$ → very general, but exponential number of param. → computationally infeasible
- independence assumption: assume pixels are independent of each other. → assump. too strong, would result in random sampling of pixels
- param. conditionals: parameterize each $p(x_i = 1 \mid x_1,\dots,x_{i-1})$ with a set of param. θ_i. total number of parameters: $\sum_i |\theta_i|$.

(FVSBN)
Fully Visible Sigmoid Belief Network: model $\hat{x}_i = p(x_i \mid x_1,\dots,x_{i-1})$ via logistic regression: $f_i(x_1,\dots,x_{i-1}) = \sigma(\alpha_i^0 + \alpha_i^1 x_1 + \dots + \alpha_i^{i-1} x_{i-1})$. ($\sigma$: sigmoid)
Then $p_{\theta_i}(x_i \mid x_1,\dots,x_{i-1}) = \text{Bernoulli}(f_i(x_1,\dots,x_{i-1}))$. parameters: $\mathcal{O}(n^2)$. $\left(\frac{n^2+n}{2}\right)$.

Neural Autoregressive Density Estimator (NADE): models binary sequences, based on MLPs. It improves on statistical and computational efficiency. Each conditional is modeled by the same NN. $W \in \mathbb{R}^{n \times d}$
→ hidden layer activations: $h_i = \sigma(b + W_{\cdot,<i} \cdot x_{<i})$ ($W_{\cdot,<i}$: first i cols of W.)
prediction: $\hat{x}_i = \sigma(c_i + V_{i,\cdot} h^{(i)})$ ($V_{i,\cdot}$: ith raw) (c: bias vector)
→ use that $h_{i+1} - h_i = W_{\cdot,i+1} \cdot x_{i+1}$ to evaluate h_{i+1} more efficiently
→ total number of Features is only $\mathcal{O}(nd)$
Training: maximize the average log-likelihood
$\frac{1}{n}\sum_{j=1}^{n} \log(p(x^{(j)})) = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{D} \log(p(x_i^{(j)} \mid x_{<i}^{(j)}))$ j: sample index.
→ efficient computation ($\mathcal{O}(T \cdot D)$), can use second-order optimizers, easily extendable to other types of observations (reals, multinomials)
→ teacher forcing: use ground truth values as input (always!) during training. At inference use own predicted values. (for conditionals)
→ random ordering of input x works just as well (during training).
Extensions:
- Real-valued NADE (RNADE): conditionals modelled by mix of Gaussians.
- DeepNADE: one Deep NN assigns cond. distr. to any var. given any subset of others.
- ConvNADE (convolutional)

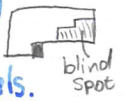
Masked Autoencoder Distribution Estimation (MADE): constructs an Autoencoder, which Fulfills the autoregressive property. → use output for $p(x_i \mid x_{<i})$ "conditionals"
→ for this must have no computational path between output unit x_d, and input units $x_d,\dots,x_D$.
→ Assign numbers $1,\dots,D$ to input units, and any value between $1, D-1$ to every hidden unit. A unit only connects to next unit (in next layer), if next unit has greater or equal value. To output layer only propagate, if value is strictly greater.
Write: $M_{ij}^{W} = 1\{m^{[l]}(i) \geq m^{[l-1]}(j)\}$, $M_{ij}^{V} = 1\{m^{[l]}(i) > m^{[l-1]}(j)\}$
- training has same complexity as other AE
- $p(x)$ is easy to compute (one forward pass)
- sampling requires D forward passes.

→ in practice, very large hidden layers necessary.

AR Generative Models of Natural Images: Decompose likelihood of image x into product of 1D distributions: $p(x) = \prod_{i=1}^{n} p(x_i | x_1, \dots, x_{i-1})$.
→problems: determine pixel ordering, model the distribution $p(x_i)$.

Pixel RNN: generate image pixels starting from one corner and model the conditionals on previous pixels with a LSTM.
→ very slow sequential generation process.
(necessary for autoregressive property)

Pixel CNN: similar approach, but model dependencies with masked convolutions.
Improve training time (rel. to PixelRNN) by using a stack of masked convolutions (which can be parallelized. But generation remains sequential.
→ Also need to be autoregressive over channels:
$p(x_i | x_1, \dots, x_{i-1}) = p(x_{i,R} | x_{<i}) \cdot p(x_{i,G} | x_{<i}, x_{i,R}) \cdot p(x_{i,B} | x_{<i}, x_{i,R}, x_{i,G})$
→Problem: stacking layers of masked convolutions creates a blind spot, i.e. in practice sometimes current pixel is not dependent on some prev. pixels.



- improvement: combine horizontal + vertical stacks of convolution. Sum outputs.
  ↳ $h_{k+1} = \tanh(W_{k,f} * h_k) \odot \sigma(W_{k,g} * h_k)$ instead of ReLU activation between the masked convolutions.
  → We call this the **Gated Pixel CNN**.

WaveNet: adapts the PixelCNN model for audio data, which has a much longer time horizon (i.e. 16.000 samples per second).
It uses dilated (stretched) convolutions to exponentially (with more layers) increase the receptive field size. →cannot use stride, bcs dim. has to be preserved.

Vector-Quantized VAE

Visual Tokenization or VQVAE: use autoencoder to convert images into a series of tokens (that are predefined).
→dim. of latent space is significantly lower, tokens more meaningful than pixels.
→ can reach state-of-the-art generation quality.

Self-Attention and Transformers: We can do sequence modeling based on the Transformer architecture. Predict based on convex combination of entire input sequence. →learn to identify relevant past information. →"Attention Operation"

- linear mapping transforms input embeddings to key, value, query embeddings.
  → $K = X \cdot W_K$, $V = X \cdot W_V$, $Q = X \cdot W_Q$, $X \in \mathbb{R}^{T \times D}$.
  → W's are learnable weights (DxD generally).

Attention mechanism: learns a dictionary representation (of X), where we map keys K to values V. Then Q identifies a set of values, which (in linear combination) form a new prediction.

- self attention: all keys, values and queries extracted from same sequence
- cross attention: queries are generated from different sequence.

vector product $a \cdot b = \|a\| \cdot \|b\| \cdot \cos\theta$

Steps: 1. Compute Q and K and check their pairwise similarity using dot product
2. obtain prob. distribution over the keys using softmax, to obtain the attention weights: $\alpha = \text{softmax}(QK^T / \sqrt{d_k}) \in \mathbb{R}^{1 \times T}$ (d_k: dim. of keys)
3. output are weighted values: $X_T = \alpha V$

→ to fulfill the autoregressive property, use mask $M = \begin{pmatrix} -\infty & \cdots & -\infty \\ 0 & \ddots & \vdots \\ & & -\infty \end{pmatrix}$ and write: $\alpha = \text{softmax}((QK^T/\sqrt{d_k}) + M)$.

quadratic ↓

⇒ expressive and powerful architectures, computationally expensive (attention $\mathcal{O}(T^2 \cdot D)$)

Change of Variables: can be used to transform a complex region into a simple one.

1D: For $x = g(u)$ $\int_{x_0=g(a)}^{x_1=g(b)} f(x)\,dx = \int_{u_0=a}^{u_1=b} f(g(u)) \cdot g'(u)\,du$.

alternatively: $z \sim p_z(z)$, $x = f(z)$ (f monotonous, diffable), $z = f^{-1}(x)$, then
$p_x(x) = p_z(f^{-1}(x)) \cdot |(f^{-1})'(x)| = p_z(z) \cdot |(f^{-1})'(x)|$. → jacobian

2D: $x = g(u,v)$, $y = h(u,v)$: $\iint_R f(x,y)\,dx\,dy = \iint_G f(g(u,v), h(u,v)) \cdot J(u,v)\,du\,dv$, $J = \begin{pmatrix} \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \end{pmatrix}$

OR $z \in \mathbb{R}^2$ random, $z = A^{-1}x$. Then $p_x(x) = p_z(z) \cdot |\det(A^{-1})|$

$f: \mathbb{R}^d \to \mathbb{R}^d$: $x = f(z)$: $p_x(x) = p_z(z) \cdot \left|\det\left(\frac{\partial f(z)}{\partial z}\right)\right|^{-1} = p_z(z) \cdot \left|\det\left(\frac{\partial f^{-1}(z)}{\partial z}\right)\right|$

↳ computationally bad (GAN's are better)

Normalizing Flows: is a latent variable (generative) model with tractable likelihood. The mapping from input X to latent variable Z is given by a deterministic and invertible $f_\theta: \mathbb{R}^d \to \mathbb{R}^d$, s.t. $X = f_\theta(Z)$, $Z = f_\theta^{-1}(X)$.

→ Then we have: $p_X(x;\theta) = p_Z(f_\theta^{-1}(x)) \cdot \left|\det\left(\frac{\partial f_\theta^{-1}(x)}{\partial x}\right)\right|$

→ f_θ can be parameterized with a NN. But the NN must be diff.able, invertible and preserve dimensionality. Efficienty computed Jacobian. (e.g. if it is triangular)

Coupling Layer: fulfills this condition.

h: element-wise fct.
β: arbitrarily complex (not necessarily invertible)

↳ forward pass: $\begin{pmatrix} y^A \\ y^B \end{pmatrix} = \begin{pmatrix} h(x^A, \beta(x^B)) \\ x^B \end{pmatrix}$

backward pass: $\begin{pmatrix} x^A \\ x^B \end{pmatrix} = \begin{pmatrix} h^{-1}(y^A, \beta(y^B)) \\ y^B \end{pmatrix}$, Jacobian: $\begin{pmatrix} h' & h'\beta' \\ 0 & 1 \end{pmatrix}$

→ Flow of transformations: One nonlinear transformation f is usually not enough. We need multiple transformations together:

$X = f(z) = f_K \circ f_{K-1} \circ \dots \circ f_1(z)$, $p_X(x) = p_Z(f^{-1}(x)) \cdot \prod_k \left|\det\left(\frac{\partial f_k^{-1}(x)}{\partial x}\right)\right|$

→ in practice, roles of A and B are switched each time. Also we randomly split A,B.

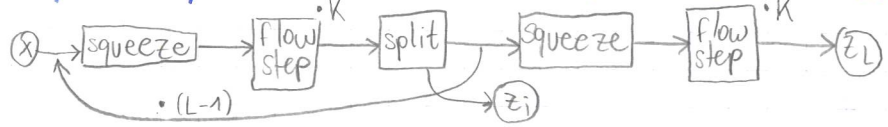
- Training: assume samples are i.i.d., maximize exact log-likelihood:

$\log(p_X(\mathcal{D})) = \sum_{x \in \mathcal{D}} \left(\log(p_Z(f^{-1}(x))) + \sum_k \log\left(\left|\det\left(\frac{\partial f_k^{-1}(x)}{\partial x}\right)\right|\right)\right)$

- Inference: • generate sample: sample $z \sim p_Z(z)$, $x = f(z)$.
  • eval. prob of x: let $z = f(x)$ and compute with $p_Z(z)$.
- intuition: through series of transformations, obtain complicated distr. of x. (step by step)

→ more transformations can model more complex distributions. A cont. NF could potentially model infinite number of transformations. (→ would also have lower comp. cost)

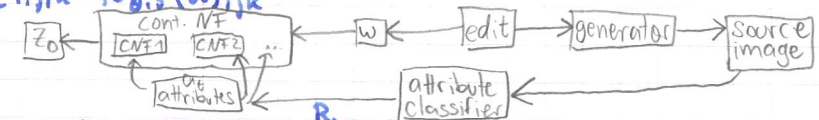
NF Architectures:



- L levels and K steps per level (depth)
- squeeze reduces spatial dim. (only pass half of features for comp. efficiency.) and also preserves the information → pass as several smaller matrices
- flow step: → actnorm → inv. 1×1 convolution → affine coupling layer →
  → activation norm: trainable (s, b) data dependent initialization
  forward: $y_{i,j} = s \odot x_{i,j} + b$, backward: $x_{i,j} = (y_{i,j} - b)/s$, log-det: $H \cdot W \cdot \text{sum}(\log|s|)$.
  → 1×1 conv: like permutation in channel dimension. take tensor h with dim. H×W×C. Init. W as random rot. matrix with $\det(W) = 1$. Efficiently compute its determinant: $W = P \cdot L \cdot (U + \text{diag}(S))$, P (fixed) permut. matrix, L is lower Δ-matrix (1 on diag), U upper Δ-matrix, S vector.
  → $\log(|\det(W)|) = \text{sum}(\log(|S|)) \to \mathcal{O}(C)$.
  → conditional coupling layer: additional input to β: $\beta(y^B; W)$

SR Flow: Get distr. of high resolution images, given a low resolution image. (we condition on it!)
$u = g_\theta(x)$ CNN encoding lr-image; activation map h^n.
Forward: $h^{n+1} = \exp(\beta^n_{\theta,s}(u)) \cdot h^n + \beta_{\theta,b}(u)$, Inverse: $h^n = \exp(-\beta^n_{\theta,s}(u)) \cdot (h^{n+1} - \beta^n_{\theta,b}(u))$
log-det: $\sum_{i,j,k} \beta^n_{\theta,s}(u)_{ijk}$

Style Flow:



→ conditions on attributes (tensors)

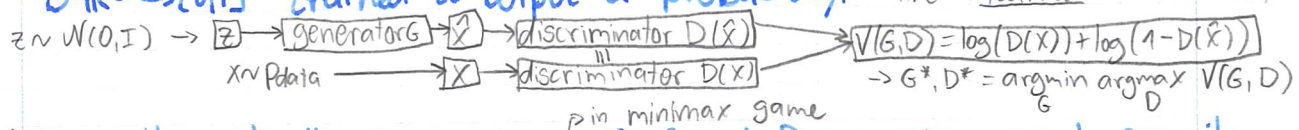
C-Flow: Use two Flows, one conditioned on the other flow A. Flow A is a standard affine coupling layer. (X^B of A is input to β of flow B).
$z_B^{\hat{}} = f_\phi^{-1}(x_B^{\hat{}} \mid x_A^{\hat{}})$, $z_A^{\hat{}} = g_\theta^{-1}(x_A^{\hat{}})$, $x_B^{\hat{}} = f_\phi(z_B^{\hat{}} \mid z_A^{\hat{}})$ → synthesize new image x_B
→ quite low resolution, and very expensive training.

Problems with likelihood learning: likelihood often bad indicator of generation quality. (of a sample)
→ can have high log-likelih. but bad samples e.g. $0.01\,p(x) + 0.99\,q(x)$ (q: noise) $\approx \log(p) - \log(100)$
→ bad log-likelih. but excellent samples, e.g. when memorizing the training data → likelihood-Base... models!

Generative Adversarial Networks (GANs): Let 2 networks play 2-player game.

Observation $x \in \mathbb{R}^D$, latent variable $z \in \mathbb{R}^Q$, $p(z)$ prior over z.

- Generator: tries to fool discriminator by generating real-looking images. → use any architecture for G, D
  $G: \mathbb{R}^Q \to \mathbb{R}^D$ maps random z to a sample from data distribution
- Discriminator: tries to distinguish between real and fake images
  $D: \mathbb{R}^D \to [0,1]$ trained to output a probability. → like a learned loss function!



↑ in minimax game

Training:
- theoretically $p_{model} = p_{data}$ if G and D are given enough capacity.
- in practice use numerical optimization (otherwise comp. expensive + overfitting). Thus do alternating optimization, i.e. k steps for D and 1 step for G to keep D near optimum and slowly changing G.

⚠ in theory!

While not converged do: 1. for k steps do: 1.1 draw N training samples $x^{(i)}$ from p_{data}, 1.2 draw N noise samples $z^{(i)}$ from $p(z)$, 1.3 update discriminator by ascending gradient: $\nabla_{\theta_D} \frac{1}{N} \sum_{i=1}^{N} \log(D(x^{(i)})) + \log(1-D(G(z^{(i)})))$

2. Draw N noise samples from $p(z)$ (i.e. $z^{(i)}$)
3. Update Generator G by descending grad.: $\nabla_{\theta_G} \frac{1}{N} \sum_{i=1}^{N} \log(1-D(G(z^{(i)})))$

⇒ GANs do not explicitly model density/likelihood. (→implicit)
- theoretical guarantees all require strong assumptions → for e.g. optimality, convergence

Derivation of GAN objective: Assuming fixed G, use binary-cross-entropy (BCE)
$\Rightarrow D^* = \text{argmin}_D -\frac{1}{2}\left(\mathbb{E}_{x \sim p_{data}}[\log(D(x))] + \mathbb{E}_{z \sim p_z}[\log(1-D(G(z)))]\right)$

for optimal G^*, define $V(G,D) = \log(D(x)) + \log(1-D(\hat{x}))$. Then we have $G^*, D^* = \text{argmin}_G \text{argmax}_D V(G,D)$.

- for any Generator G, the optimal discriminator is $D^* = \frac{p_{data}(x)}{p_{data}(x) + p_{model}(x)}$
  proof: maximize $\mathbb{E}_x[V(G,D)]$ wrt. x (p_{model} and p_{data})

same as here ↓

Jensen-Shannon Divergence: similarity metric for distributions (like D_{KL})

$$D_{JS}(p\|q) = \frac{1}{2} D_{KL}\left(p \,\|\, \frac{p+q}{2}\right) + \frac{1}{2} D_{KL}\left(q \,\|\, \frac{p+q}{2}\right)$$

Global optimum: of training criterion: $V(G,D^*) = \mathbb{E}_{x \sim p_{data}}[\log(D^*(x))] + \mathbb{E}_{x \sim p_{model}}[\log(1-D^*(x))]$ is achieved iff $p_{data}(x) = p_{model}(x)$ and $V(G,D^*) = -\log(4)$. → $-\log(4) + 2 D_{JS}(p\|p_m)$ strong
Proof: follows with D_{JS} and that $D_{JS} \geq 0$ and $=0$ iff $p=q$ ($D_{JS}(p\|q)$) ↗ assump.

GAN Convergence: If D, G have enough capacity, at each update step D reaches D^* and p_{model} is updated to improve: $V(p_{model}, D^*) = \mathbb{E}_{data}[\log(D^*(x))] + \mathbb{E}_{model}[\log(1-D^*)]$
Then p_{model} converges to p_{data}.

GAN training in practice: opt. $\log(1-D(G(z)))$ can saturate easily during beginning of training, when G is still bad. Thus, in step 3 of the training pseudocode (above), instead train G to maximize $\log(D(G(z)))$ → "non-saturating loss" → same fixed points, much stronger grads in early learning.
↗ now ascend instead of descent

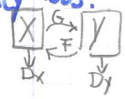
GAN Problems:
- mode collapse: generator G only learns to produce high-quality sample with very low variability. → cover only fraction of $p_{data}(x)$.
  ↳ iteratively it collapses into different modes
  → solution: use unrolled GANs, after k updates of D, update G with the D of after the next k steps. → discourages G to exploit local minima. → make G prepared for next moves.
- training instability: must find Nash Equilibrium. Bad progress for one can be good for the other.
- JS-Divergence: high-dim. space of p_{data} and p_{model}. They (support) might not overlap. Significantly slower convergence (zero gradients)
  → solution: use different prob. distance metric: Wasserstein distance.
- stabilize gradient with grad penalty:
  $V(G,D) = \mathbb{E}_{data}[\log(D(x))] - \lambda \|\nabla_x D(x)\|^2] + \mathbb{E}_{model}[\log(1-D(x))]$.

Deep Convolutional GANs (DCGAN): Replace any pooling layers with strided conv. (in D) and fractional-strided conv. (in G). Use batchnorm in G and D. Remove fully connected hidden layers for deeper architectures. Use ReLU activation in G in all layers, except output where tanh. Use Leaky ReLU activation in D for all layers.

GAN discussion: + very flexible model, + only backpropagation required for training (no sampling), + no approx. of likelihood necessary (like in VAEs) + very good in practice performance. (state of the art in e.g. image-to-image) - no explicit p_{model}, - no direct way to evaluate sample likelihood, - training requires careful balancing of G and D. (- D overfits easily with small dataset → use data augmentation)

Pix2Pix: Image-to-image translation with conditional GANs. Loss Fct. is
$L(G,D) = \underbrace{\mathbb{E}_{x,y}[\log(D(x,y))] + \mathbb{E}_{x,z}[1-\log(D(x,G(x,z)))]}_{L_{cGAN}(G,D)} + \lambda \cdot \underbrace{\mathbb{E}_{x,y,z}[\|y-G(x,z)\|_1]}_{L_{L1}(G)}$
→ requires paired images as training data (limitation)
→ maps into unique output, but many valid outputs exist

CycleGAN: tries to do unpaired image translation. Uses a cycle consistency loss.
$L(G,F,D_X,D_Y) = L_{GAN}(G,D_Y,X,Y) + L_{GAN}(F,D_X,Y,X) + \lambda \cdot L_{cyc}(G,F)$
with $L_{cyc}(G,F) = \mathbb{E}_{x\sim p_{data}}[\|F(G(x))-x\|_1] + \mathbb{E}_{y\sim p_{data}}[\|G(F(y))-y\|_1]$



Vid2Vid: $L_{cGAN}(G,D) = \mathbb{E}_{x,y}[\log(D(x,y))] + \mathbb{E}_x[1-\log(D(x,G(x)))]$
Sequential generator: $P(\tilde{y}_{1:T}|x_{1:T}) = \prod_{t=1}^{T} p(\tilde{y}_t | \tilde{y}_{t-L:t-1}, x_{t-L,t})$

Diffusion models: gradually reduce/add noise from images. This results in higher quality generation, more diversity, more stable training (compared to GAN).
- two directions: denoising (is learned), diffusion (deterministic) → $p_\theta(x_{t-1}|x_t)$ and $q(x_t|x_{t-1})$

→ model as a Markov Stochastic Process with T steps, where $x_0 \sim q(x_0)$ (= p_{data}) and $x_T \sim \mathcal{N}(0,I)$ a random sample. (i.e. larger t → noisier image)
- **Diffusion Step:** variance schedule determines how much noise is introduced at each step. $\{\beta \in (0,1)\}_{t=1}^{T}$ s.t. $0<\beta_1<\dots<\beta_T<1$ and $q(x_t|x_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}\cdot x_{t-1}, \beta_t I)$. Note $x_t = \sqrt{1-\beta_t}\, x_{t-1} + \sqrt{\beta_t}\,\varepsilon^{\sim\mathcal{N}(0,I)}$ → reparametrization trick
→ For $\bar{\alpha}_t = \prod_{i=1}^{t}\alpha_i = \prod_{i=1}^{t}(1-\beta_t)$, $q(x_t|x_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}\,x_0, (1-\bar{\alpha}_t)\cdot I)$. → big speedup
- **Denoising Process:** model each denoising step with the same NN, which then gives $p_\theta(x_{t-1}|x_t)$, which approximates $q(x_{t-1}|x_t, x_0)$. Write $p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t,t), \sigma_t \cdot I) \approx q(x_{t-1}|x_t,x_0)$.

ELBO loss: $L = \underbrace{\mathbb{E}_{q(x_1|x_0)}[\log(p_\theta(x_0|x_1))]}_{L_0} - \underbrace{D_{KL}(q(x_T|x_0)\|p(x_T))}_{L_T} - \sum_{t=2}^{T}\underbrace{\mathbb{E}_{q(x_t|x_0)}[D_{KL}(q(x_{t-1}|x_t,x_0)\|p_\theta(x_{t-1}|x_t))]}_{L_t}$

- objective function: since $p_\theta(x_{t-1}|x_t)$ and $q(x_{t-1}|x_t,x_0)$ are Gaussian, we have $\operatorname{argmin}_\theta D_{KL}(q(x_{t-1}|x_t,x_0)\|p_\theta(x_{t-1}|x_t)) = \operatorname{argmin}_\theta \frac{1}{2\sigma_q^2(t)}\cdot(\|\mu_\theta - \mu_q\|_2^2)$

→ optimize to match $\mu_\theta(x_t,t)$ and $\mu_q(x_t,x_0)$. We can write both as follows $\mu_q(x_t,x_0) = \frac{1}{\sqrt{\alpha_t}}x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}\sqrt{\alpha_t}}\varepsilon_0$, $\mu_\theta(x_t,t) = \frac{1}{\sqrt{\alpha_t}}x_t - \frac{1-\alpha_\theta}{\sqrt{1-\bar{\alpha}_t}\sqrt{\alpha_t}}\cdot\hat{\varepsilon}_\theta(x_t,t)$ ($\varepsilon_0 \sim \mathcal{N}(0,I)$)
→ in practice, trying to predict ε_0 with a NN $\hat{\varepsilon}_\theta(x_t,t)$ works much better than trying to predict x_0 or $\mu_q(x_t,x_0)$ directly.

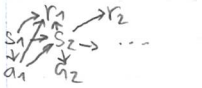
- Pseudocodes.
Training: Repeat until converged: 1. $x_0 \sim q(x_0)$ ($\sim p_{data}$), 2. $t \sim \text{Unif}(\{1,\dots,T\})$, 3. $\varepsilon \sim \mathcal{N}(0,I)$, 4. Gradient step (descent): $\nabla_\theta \|\varepsilon - \varepsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon, t)\|^2$.
Sampling: 1. $x_T \sim \mathcal{N}(0,I)$, 2. for $t=T,\dots,1$ do: 2.1. $z\sim\mathcal{N}(0,I)$ if $t>1$ else $z=0$, 2.2. $x_{t-1} = 1/\sqrt{\alpha_t}(x_t - (1-\alpha_t)/\sqrt{1-\bar{\alpha}_t}\cdot\varepsilon_\theta(x_t,t)) + \sigma_t z$ (where $\sigma_t^2 = \beta_t$), 3. ret. x_0.
→ $\varepsilon_\theta(x_t,t)$ is typically a U-net, and shared across timesteps (time t is input!).
→ There exist continuos-time diffusion models, with infinite diffusion steps. This allows to define a unified formulation for all diffusion models under the lens of stochastic/ordinary PDEs.

Conditional generation with diffusion models: Want to introduce explicit control over generated data, using conditionals, i.e. we want to find

$p_\theta(x_{0:T}|y) = p(x_T) \cdot \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t, y)$ (→ y could be eg. text or image.)

Diffusion Guidance: trade diversity for more quality, to align the generations with the conditioning: $\varepsilon^*_\theta(x_t, c, t) = (1+\rho)\varepsilon_\theta(x_t, c, t) - \rho \cdot \varepsilon_\theta(x_t, t)$ ↳ Unconditional

Latent Diffusion Models: Compress input into latent space, then perform diffusion/denoising in latent space. → because regular DiffM are comp. expensive.
→ Significantly more efficient. Diffusion can focus on image semantics. → state of the art approach. (Dall-E).



Markov Decision Process (MDP): is (S, A, R, P, γ), States S, actions A, reward function $r: S \times A \to \mathbb{R}$, transition fct. $p: S \times A \to S$, $s_0 \in S$ init. state, discount factor γ. → Describes the Reinforcement Learning setting.

Policy and Value function: $\pi: S \to A$ describes how we act. This induces a value function $V_\pi: S \to \mathbb{R}$, the expected cumulative reward.

Bellman equation for V_π: describes a recursive dependency of V_π (to itself)

$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ then $V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma \cdot G_{t+1} | S_t = s]$

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

$\Rightarrow V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S') | S_t = s]$. → non-linear, no closed-form solution.
→ can be solved with Dynamic Programming, Monte Carlo Methods, TD-Learning.

Dynamic programming: requires MDP with transition probabilities. Limited utility, but great theoretical importance. We focus on discrete space.

- policy iteration: estimates π^*. 1. Compute V_π for an arbitrary π, 2. Update π, given $V_\pi \Rightarrow \pi'$, 3. Iterate until $\pi \approx \pi'$.
- Value iteration: estimates π^*. Compute optimal V^*, then obtain π^* from V^*.
  - greedy policy based on V_π: $\pi'(s) = \text{argmax}_{a \in A}(r(s,a) + \gamma V_\pi(p(s,a))) \Rightarrow V_{\pi'} \geq V_\pi$.
  → update step: $V^*_{new}(s) = \max_{a \in A}(r(s,a) + \gamma V^*_{old}(s'))$
  → the greedy policy of optimal V^* is π^* itself.

  Pseudocode: 1. Init. $i=0$, $V_0 = 0$, 2. while $\delta > \theta$: 2.1. $\delta \leftarrow 0$, 2.2 For all $s \in S$ do: 2.2.1. $v \leftarrow V(s)$, 2.2.2. $V(s) = \max_{a \in A} \sum_{s',r} p(s', r|s, a) \cdot (r + \gamma V(s'))$, 2.2.3. $\delta \leftarrow \max\{\delta, |v - V(s)|\}$, 3. $\pi_i \leftarrow$ greedy policy from V_i.

⊕ → Pro: guaranteed to converge in finite iterations, VI more efficient than PI ↗ typically
⊖ → Cons: Need prob. transition matrix, iterate over whole state space, much memory.

tabular

Monte Carlo Sampling: runs episodes of a policy π and compute samples of the term in the expectation of $V_\pi(s_t) = \mathbb{E}_{a_t \sim \pi, s_{t+1} \sim p(s_t, a_t)}[\sum_{i=0} \gamma^i \cdot r(s_{t+i}, a_{t+i})]$

⊕ → unbiased estimate, and its not necessary to know system dynamics.
⊖ → high variance, exploration/exploitation dilemma, slow if episodes are long → needs termination state

Temporal-Difference (TD-) Learning: After each step with our policy, we compute difference to our estimate, and update the value function V. → only update visited states.

→ $\Delta V(s) = r(s,a) + \gamma \cdot V(s') - V(s)$, $V(s) \leftarrow V(s) + \alpha \cdot \Delta V(s)$. ($\alpha$: learning rate)

- SARSA: on-policy, i.e. computes Q-value acc. to a policy, then agent follows that policy. Instead, update Q-fct.: $\Delta Q(S,A) = R + \gamma Q(S', A') - Q(S, A)$, $Q(S,A) \leftarrow Q(S,A) + \alpha \Delta Q(S,A)$
- Q-Learning: off-policy, i.e. comp. Q-val acc. to greedy update, but agent follows different exploration policy. Update: $\Delta Q(S,A) = R_{t+1} + \gamma \cdot \max_{a \in A}(Q(S', a)) - Q(S, A)$

⊕ → less variance than MC-sampling (due to bootstrapping), more sample efficient than DP, don't need to know transition prob. matrix.
⊖ → biased (due to bootstrapping), exploration/exploitation dilemma.

Deep Reinforcement Learning (Deep RL): Prior, only tabular setting. Now, we extend the ideas to arbitrarily large state spaces (function approximation). Also, replace action-val-methods with parametrized policies → policy based gradient methods

- **Deep Q-learning:** use NN to approximate $Q_\theta(s,a)$. Apply SGD to minimize the loss $L(\theta) = (R + \gamma \cdot \max_{a \in A}(Q_\theta(s', a')) - Q_\theta(s, A))^2$ → no backprop. through max-term
  → randomly sample from replay buffer to fulfill SGD i.i.d samples requirement. Otherwise samples from same trajectory are strongly correlated.
  → can achieve superhuman performance

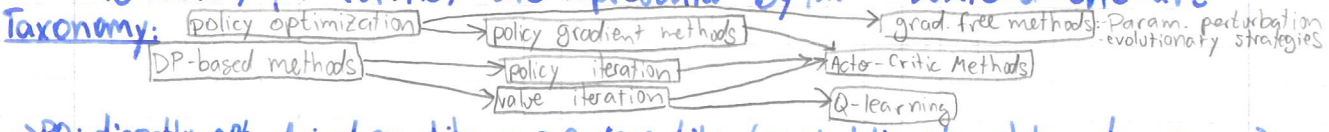
Policy Search methods: directly learn policy, instead of value function. Let the policy be a stochastic function approximator $\pi_\theta(a_t|s_t) = \mathcal{N}(\mu_t, \sigma_t^2|s_t)$, which can be modeled by a NN. trajectory: $p(\tau) = p(s_1)\prod_{t=1}^{T}\pi(a_t|s_t)\,p(s_{t+1}|a_t,s_t)$

- policy update: maximize expected trajectory reward via SGD find $\theta^* = \arg\max J(\theta)$ → gradient ascend $J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \gamma^t \cdot r(s_t,a_t)\right]$

→ $\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p(\tau)}\left[\left(\sum_{t=0}^{T}\nabla_\theta \log(\pi_\theta(a_t^i|s_t^i))\right)\cdot\left(\sum_{t=0}^{T}\gamma^t \cdot r(s_t^i,a_t^i)\right)\right]$ (baseline: $-b(s_t^i)$) → how to sample this?

- REINFORCE: use sample average (MC sampling) to compute gradient → high variance. Introduce any baseline fct. to reduce variance, $b(s_t^i)$ → avg. reward, est. V-fct. (examples) → stays unbiased $\nabla_\theta J(\theta)$
- Actor-Critic methods: use bootstrapping for estimating rewards. → func. approx. instead of traj. rollouts.
  → $\nabla_\theta J(\theta) = 1/N \sum_i \sum_{t=0}^{T}\nabla_\theta \log(\pi_\theta(a_t^i, s_t^i))\cdot(r(s_t^i,a_t^i) + \gamma V(s_{t+1}^i) - V(s_t^i))$
  → π_θ (actor), V (critic) are represented by NN → state-of-the-art

Taxonomy:



→ PO: directly opt. desired quantity, more versatile (cont./discrete state-action spaces)
→ DP: indirect, exploit system structure, self consistent, more compatible with off-policy and exploration, more sample efficient, mostly only discrete spaces.

Deep Mimic: learns physics-based motion policies, tries imitating reference motions state $s_t \triangleq (q_t, \dot{q}_t, \hat{q}_{t+1})$, reward $r_t = w^I \cdot r_t^I + w^G \cdot r_t^G$, r_t^I = reference motion − executed motion, r_t^G = task objective. Model policy as gaussian $\pi(a|s,g) \sim \mathcal{N}(\mu(s), \Sigma)$, learn $\mu(s)$, fix Σ.

3D Representation of objects: for modelling in 3D-space. → For 3D-surfaces.

- Voxels discretize 3D space into 3D grid. $\mathcal{O}(n^3)$ memory, but only limited resolution
- Points / little spheres: discretization of surface. Does not model connectivity / topology.
- Meshes: discretization into vertices and faces (like graph), granularity, requires class-specific template, max num. of vertices, leads to self-intersections (error).
- (Neural) Implicit Surface Repoesentation: perfectly represent surface as level-set of a continuous function f. Very exact, but no direct graphical visualization. Represent f either with occupancy network $f_\theta: \mathbb{R}^3 \times X \to [0,1]$ prob. of being inside surface, or DeepSDF (signed distance field): $f_\theta: \mathbb{R}^3 \times X \to \mathbb{R}$ distance from surface (inside: negative, outside: positive).

→ Implementation: parametrize f_θ as MLP. Condition over type of shape via concatenating the input. For training we represent the ground truth as watertight meshes, Point clouds, or 2D images.

- watertight mesh: randomly sample points in space, train on binary cross entropy. (↗ inside/outside of surface. (?)
- point clouds: direct output of many sensors, learn f_θ signed distance. With min. loss $L(\theta) = \sum_i |f_\theta(x_i)|^2 + \lambda\,\mathbb{E}_x[(\|\nabla_x f_\theta(x)\| - 1)^2]$ → vanish term + λ Eikonal term (↳ smooth plane)
- 2D-images: as training input. We need to render this in a differentable way. DVR (diff.able volumetric rendering) describes learning f_θ (occupancy function) and t_θ (texture) from a 2D image. Forward pass: r_0 pos. of camera, u a pixel, consider w vector connecting r_0 to u, and ray $r(d) = r_0 + wd$. $\hat{p}$ is first point of intersection with surface ($f_\theta(\hat{p}) = \tau$, found with secant method), $\hat{d}$ is distance from r_0 to $\hat{p}$: $r(\hat{d}) = \hat{p}$, obtain $t_\theta(\hat{p})$, color pixel u with color $t_\theta(\hat{p})$ (↗ connect with line → root, $\frac{x_1 - x_0}{f(x_1) - f(x_0)}$ $(x_0, f(x_0)), (x_1, f(x_1))$)

(secant rule: apply iteratively, to find a zero of the fct: root $x_2 = x_1 - f(x_1)$)

Backward pass: I is image, loss is $L(I,\hat{I}) = \sum_u \|\hat{I}_u - I_u\|$, we receive the gradient $\frac{\partial L}{\partial \theta} = \sum_u \frac{\partial L}{\partial \hat{I}_u}\frac{\partial \hat{I}_u}{\partial \theta}$ with $\frac{\partial \hat{I}_u}{\partial \theta} = \frac{\partial t_\theta(\hat{p})}{\partial \theta} + \frac{\partial t_\theta(\hat{p})}{\partial \hat{p}}\cdot\frac{\partial \hat{p}}{\partial \theta}$. Further note

$f_\theta(\hat{p}) = \tau \xrightarrow[\text{differentation}]{\text{implicit (wrt. }\theta\text{)}} 0 = \frac{\partial f_\theta(\hat{p})}{\partial \theta} + \frac{\partial f_\theta(\hat{p})}{\partial \hat{p}}\frac{\partial \hat{p}}{\partial \theta} \Rightarrow \frac{\partial \hat{p}}{\partial \theta} = -w\left(\frac{\partial f_\theta(\hat{p})}{\partial \hat{p}}\cdot w\right)^{-1}\frac{\partial f_\theta(\hat{p})}{\partial \theta}$ ($\hat{p} = r_0 + \hat{d}w$)

→ Analytic solution, no need to store intermediate results.

Neural Radiance Field (NeRF): model 3D scene from 2D input images. Introduces the concept of density σ to predict along with the color c.
→ input x,y,z and first predict σ, then predict with σ, and the camera angles Φ,Θ, the final RGB value of pixel. For σ, re-inject x,y,z into middle layer.
Procedure: sample points along whole ray from camera r_0 to pixel u (→ don't stop at first intersection with surface, like before: 2D-images)

↳ [Neural Radiance Field - Procedure]: Final color of a pixel is computed as weighted average of colors along the ray. Let t_i be the i-th sampled point, $\delta_i = t_{i+1} - t_i$, $\alpha_i = 1 - \exp(-\sigma_i \delta_i)$, transmittancy $T_i = \prod_{j=1}^{i-1} (1-\alpha_i)$, final color $C = \sum_i T_i \alpha_i c_i$.
→ more efficient: sample more where weights ($T_i \alpha_i$) are higher.
→ Minimize reconstruction error: $\min_\theta \sum_i \| \text{render}_i(F_\theta) - I_i \|^2$
• flexible: can model transparency / thin structures, but usually bad geometry
• Positional Encoding: solves difficulty with representing high-frequency variation in color and geometry. Pass input to higher-dim. fourier feature space $\sin(2^k \pi p), \cos(2^k \pi p)$
→ NeRF: slow, needs much (50+ views) data, only static scenes
→ maybe try to only render for small cubes/spheres/ellipsoids → problems with thin structures.
Gaussian Splatting: uses 3D Gaussians to model a surface. For rendering, project these Gaussians into 2D space. Calculate opacity: $\alpha = \sigma \cdot \exp(-\frac{1}{2}(x-\mu')^T \Sigma'^{-1} (x-\mu'))$ ↳ of 2D Gaussian
→ final color: $C = \sum_i T_i \alpha_i c_i$

2D human pose representation: model with k-vertices graph (G,V). Let $L_i = (x_i, y_i)$ be the given estimate for vertex i in the 2D image I. We define the score fct. $S(I,L) = \sum_i \alpha_i^{m_i} \cdot \phi(I, l_i) + \sum_{i,j} \beta_{ij}^{m_i m_j} \psi(l_i, l_j)$, ϕ is unary term measures distance (to correct l_i), ψ measures deformation between two vertices i,j. (m_i: mixture type)
↳ usually also add co-occurrence bias $S(M) = \sum_{ij} b_{ij}^{m_i m_j}$, where b is the pairwise co-occurrence prior between part i with mixture m_i, and j with m_j.
→ Want to maximize $S(I,L)$. ($L = (l_1, \dots, l_k)$.)
• Direct Regression: to learn representation. Directly regress x and y coord., using Deep CNNs and a refiner.
• Heatmaps: to learn representation. Based on DeepCNN. Choose keypoints and create separate heatmaps (Gauss. distr.) for each. (or binary)
3D human pose representation: SMPL (skinned multi-person Linear model) uses 3D mesh with 7000 vertices as representation of human. It is difficult to get mesh from a raw scan. Do PCA on different meshes in canonical pose to estimate direction of maximal shape variation → obtain low dim (10-300) subspace (↗ to define a pose)
• Linear Blend Skinning (LBS): Posed vertices are linear combination of transformed template vertices $t_i' = \sum_k w_{ki} G_k(\theta, J) \cdot t_i$, t transformed vertices, w blend skinning weights, G_k rigid bone transformation, θ pose, J joint locations
→ fast and simple to compute, but well known artifacts (errors).
• SMPL: $t_i' = \sum_k w_{ki} G_k(\theta, J(\beta)) \cdot (t_i + S_i(\beta) + P_i(\theta))$ → joints J depend on shape β
↳ 1. define base template mesh, 2. capture raw training scans, 3. assign mesh templat, 4. define shape in canonical pose, 5. Factor body shape variation from pose, 6. learn pose-dependent deformations, 7. pose the mesh using linear blend skinning (LBS), 8. learn everything (including the blend weights)
→ finally obtain mesh $M(\beta, \theta, \phi)$ (shape, pose, gender), joints position $J(\beta, J, \bar{T}, S)$.
• Learned Gradient Descent: when regressing from image to 3D human model. Let a NN learn the gradient descent step: $\theta^{t+1} = \theta^t + F_w(\frac{\partial L_{\text{repro},i}}{\partial \theta}, \theta^t, X)$ (NN; gradients; current state; ← target 2D pose)

Challenges - from key points to detailed 3D surfaces: Self-occlusion, lack of depth information, articulated motion, non-rigid deformation.
• template based capture: can help, but much preprocessing, no public access, automatic and general pipeline required.
• regression based, combined with template based works well.
↳ flexible, memory efficient, generalizes to shape/pose/clothing.