- Derivatives of common functions:
  $\sigma'(x) = \sigma(x)(1-\sigma(x))$, $\tanh'(x) = 1 - \tanh^2(x)$, $|x|' = x/|x|$.
- If $\boldsymbol{x} \in \mathbb{R}^{m_1 \times \cdots \times m_d}$ and $\boldsymbol{y} \in \mathbb{R}^{n_1 \times \cdots \times n_\ell}$, then
$$\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{y}} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_\ell}.$$
- When computing gradient, think about changing parameters and how that would affect the output.
- Always type-check gradients.
- Quotient rule: $(f/g)'(x) = f'(x) \cdot g(x) - f(x) \cdot g'(x) / g^2(x)$.
- $\frac{\partial \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{A}$, $\frac{\partial \boldsymbol{x} \odot \boldsymbol{y}}{\partial \boldsymbol{x}} = \operatorname{diag}(\boldsymbol{y})$, $\frac{\partial \boldsymbol{x} \odot \boldsymbol{y}}{\partial \boldsymbol{y}} = \operatorname{diag}(\boldsymbol{x})$.

Neural networks

Perceptron: The original perceptron was a single-layer perceptron with non-linearity $\mathbb{1}\{x > 0\}$. Classification is then done by $\hat{y} = \mathbb{1}\{\boldsymbol{w}^\top \boldsymbol{x} + b > 0\}$. The learning algorithm iteratively applies $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta(y_i - \hat{y}_i)\boldsymbol{x}_i$.

If the data is linearly separable, the perceptron converges in linear time.

MLP: $\hat{y} = \sigma(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \cdots \sigma(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) \cdots + \boldsymbol{b}_{k-1}) + \boldsymbol{b}_k)$.

Do not forget biases!

Loss functions:

- MLE: $\operatorname{argmin}_{\boldsymbol{\theta}} -\sum_{i=1}^n \log p(y_i \mid \boldsymbol{\theta})$.
- BCE (MLE with Bern.): $\operatorname{argmin}_{\boldsymbol{\theta}} -\sum_{i=1}^n y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)$.
- MAP: $\operatorname{argmin}_{\boldsymbol{\theta}} -\log p(\boldsymbol{\theta}) - \sum_{i=1}^n \log p(y_i \mid \boldsymbol{\theta})$.

Early stopping: Stop training if the validation error has increased for the last p checks. We check every n epochs.

Backpropagation: Linear-time algorithm to compute gradients using chain rule. Intuitively, gradients work well for updating weights, because it measures the direction in which the loss decreases at a point. Gradient descent update rule:
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}).$$

Universal approximation theorem: Let g be any function on unit hypercube. Let $\epsilon > 0$, then there exists a NN with a single hidden layer that can approximate this function with ϵ precision,
$$|f_{\boldsymbol{\theta}}(\boldsymbol{x}) - g(\boldsymbol{x})| < \epsilon, \quad \forall \boldsymbol{x} \in I_m.$$
In words: An MLP with a single hidden layer and continuous non-linear activation function can approximate any continuous function with arbitrary precision.

Convolutional neural networks

Convolution: $(\boldsymbol{K} * \boldsymbol{I})[i,j] = \sum_{m=-k}^k \sum_{n=-k}^k \boldsymbol{K}[m,n]\boldsymbol{I}[i-m, j-n]$. Correlation uses plus instead of minus. Convolution is commutative, linear, and shift-equivariant. Can be implemented as matrix operation:
$$\boldsymbol{K} * \boldsymbol{I} = \begin{bmatrix} k_1 & 0 & \cdots & 0 \\ k_2 & k_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & k_m \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_n \end{bmatrix}.$$

CNN: Sequence of alternating convolutional and pooling layers. Convolutional layer from C_{in} to C_{out} channels:
$$\boldsymbol{Z}_j^{(\ell)} = \sum_{k=1}^{C_{\text{in}}} \boldsymbol{W}_{kj}^{(\ell)} * \boldsymbol{Z}_k^{(\ell)} + b_j, \quad j \in [C_{\text{out}}].$$
With $C_{\text{in}} = C_{\text{out}} = 1$, we have derivative:
$$\delta^{(\ell-1)}[i,j] \doteq \frac{\partial \mathcal{L}}{\partial z^{(\ell-1)}[i,j]} = \sum_{i'} \sum_{j'} \delta^{(\ell)}[i',j'] w^{(\ell)}[i'-i, j'-j]$$
$$\boldsymbol{\Delta}^{(\ell-1)} = \boldsymbol{\Delta}^{(\ell)} * \operatorname{Rot}_{180}(\boldsymbol{W}^{(\ell)}), \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(\ell)}} = \boldsymbol{\Delta}^{(\ell)} * \boldsymbol{Z}^{(\ell-1)}.$$
$D_{\text{out}} = \left\lfloor \frac{D_{\text{in}} + 2 \times p - d \times (k-1) - 1}{s} + 1 \right\rfloor$. Params: $(C_{\text{in}} \times H \times W + 1) \times C_{\text{out}}$.

Max-pooling layer:
$$z^{(\ell)}[i,j] = \max\Big\{ z^{(\ell-1)}[i',j'] \mid i' \in [i : i+k], j' \in [j : j+k] \Big\}$$
$$\frac{\partial z^{(\ell)}[i,j]}{\partial z^{(\ell-1)}[i',j']} = \mathbb{1}\{[i',j'] = [i^\star, j^\star]\}.$$
No learnable parameters, only propagation of the error.

Fully convolutional neural networks

Downsample, then upsample. *Examples*: Semantic segmentation, Image-to-image translation, Human pose estimation.

Upsampling methods:

- Nearest neighbor: Put value into all corresponding cells.
- Bed of nails: Only put value into the top-left cell.
- Max unpooling: Remember original position from max-pooling.
- Transposed convolution: Insert $s-1$ zeros between pixels and $k-p-1$ zeros as padding. Then, convolve with kernel.

U-net: Add skip-connection between corresponding down- and upsampling layers. This facilitates the combination of global from skip-connection with local information from previous layer.

Recurrent neural networks

Processes sequential data and is able to take variable-length input. At each timestep, use the same network:
$$\boldsymbol{h}^{(t)} = f_{\boldsymbol{\theta}}\Big(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\Big).$$
This behaves like a dynamical system.

Elman RNN: $f_{\boldsymbol{\theta}}(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}) = \tanh\big(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)}\big)$. $\boldsymbol{h}^{(t)}$ represents the sequence until timestep t, which we can use as input into an MLP for further processing.

Backpropagation through time (BPTT) to optimize by unrolling the RNN and applying backpropagation on the computational graph:
$$\frac{\partial \ell^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{k=1}^t \frac{\partial \ell^{(t)}}{\partial \hat{\boldsymbol{y}}^{(t)}} \frac{\partial \hat{\boldsymbol{y}}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial^+ \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}_h}.$$
We have
$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} = \prod_{i=k+1}^t \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}.$$
Suffers from exploding or vanishing gradient because of the many multiplications of $\boldsymbol{W}_h$ with itself in the gradient. If the largest eigenvalue of this matrix is greater than the upper bound of the non-linearity, the gradient will explode. If it is smaller, it will vanish.

Leaky unit: Make sure there is constant error flow to solve vanishing gradient problem:
$$\hat{\boldsymbol{h}}^{(t)} = f_{\boldsymbol{\theta}}\Big(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\Big)$$
$$\boldsymbol{h}^{(t)} = \alpha \boldsymbol{h}^{(t-1)} + (1-\alpha)\hat{\boldsymbol{h}}^{(t)}.$$

LSTM: Take idea of leaky unit to the next level by introducing gates, which protect the memory cell to make sure there is always error flow:
$$\boldsymbol{f}^{(t)} = \sigma\Big(\boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xf}\boldsymbol{x}^{(t)}\Big)$$
$$\boldsymbol{i}^{(t)} = \sigma\Big(\boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xi}\boldsymbol{x}^{(t)}\Big)$$
$$\boldsymbol{o}^{(t)} = \sigma\Big(\boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xo}\boldsymbol{x}^{(t)}\Big)$$
$$\boldsymbol{g}^{(t)} = \tanh\Big(\boldsymbol{W}_{hg}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xg}\boldsymbol{x}^{(t)}\Big).$$
Then, the memory cell and hidden state are computed by
$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \boldsymbol{g}^{(t)}$$
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh\Big(\boldsymbol{c}^{(t)}\Big).$$
The *forget gate* $\boldsymbol{f}$ decides what information to keep from previous cell state. The *gate gate* $\boldsymbol{g}$ decides what to write to the cell state. The *input gate* $\boldsymbol{i}$ decides what values of the cell state should be updated. The *output gate* $\boldsymbol{o}$ decides what values of the cell state to put into the hidden state.

The gates form an "information highway" that can easily propagate errors through the cell state due to the minimal modifications made to it.

Gradient clipping: Solve exploding gradient by limiting the norm of the gradient,
$$\boldsymbol{\theta} \leftarrow \begin{cases} \boldsymbol{\theta} - \eta \boldsymbol{g}, & \|\boldsymbol{g}\| \le k \\ \boldsymbol{\theta} - \eta \frac{k}{\|\boldsymbol{g}\|} \boldsymbol{g}, & \text{otherwise.} \end{cases}$$

Autoencoders

Autoencoders are generative models, meaning that they model the underlying distribution of the data, which makes it possible to sample from it. It works by an encoder-decoder structure, where f maps data points $\boldsymbol{x} \in \mathbb{R}^n$ to latent variables $\boldsymbol{z} \in \mathbb{R}^d$, i.e., encodes the information compactly in $d \ll n$ dimensions. The decoder g maps latent variables $\boldsymbol{z}$ back to the input space for a reconstruction $\hat{\boldsymbol{x}}$. Thus, $g \circ f$ aims to approximate the identity function. The assumption is that if the decoder is able to reconstruct the original input from the latent representation, this representation must be meaningful.

Linear autoencoder: Principal component analysis.

Non-linear: Parametrize f and g as NNs to gain performance.

VAE: Autoencoders have bad sampling quality, because the latent space is not well-structured, meaning that there is no continuity or interpolation. The reason for this is that there are large regions in the latent space where there are no observations.

The solution is to make the generator output a distribution over latents. Specifically, it outputs a Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}), \operatorname{diag}(\boldsymbol{\sigma}^2_{\boldsymbol{\theta}}(\boldsymbol{x})))$ over latent vectors. However, naively using this method will result in very different $\boldsymbol{\mu}$ with very low $\boldsymbol{\sigma}^2$ for the different data points, which is essentially the same as outputting points.

To solve this, we must minimize the KL-divergence between the output distribution and the standard Gaussian. This encourages the encoder to distribute the encodings evenly around the origin.

We want to maximize the likelihood $p(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})p(\boldsymbol{z})\mathrm{d}\boldsymbol{z}$. However, this is intractable. The best we can do is optimize the ELBO:

$$\log p(\boldsymbol{x}) \geq \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\theta}}(\cdot\mid\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})] - \mathrm{KL}(q_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}) \| p(\boldsymbol{z})).$$

We need to use the reparametrization trick to take the gradient of the expectation, which means that instead of sampling $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}, \operatorname{diag}(\boldsymbol{\sigma}^2))$, we sample $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and compute $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$. After training, we can sample from the distribution by sampling $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and giving this to the decoder.

β-**VAE**: The VAE still has problems with its latent space: It is *entangled*. A latent space is disentangled if each dimensions changes a single feature of the output. We solve this by introducing a hyperparameter β which gives more weight to the KL term. The intuition behind this is that if factors are in practice independent from each other, the model should benefit from disentangling them. This can be derived theoretically as a Lagrangian.

Autoregressive models

Autoregressive models can compute the likelihood $p(\boldsymbol{x})$ in a tractable way by the chain rule:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i \mid \boldsymbol{x}_{1:i-1}).$$

The hard part of this approach is that we must parametrize all possible conditional distributions $p(x_{k+1}|\boldsymbol{x}_{1:k})$.

FVSBN: Fully visible sigmoid belief networks parametrize each timestep by its own function:

$$f_i(\boldsymbol{x}_{1:i-1}) = \sigma\Big(\alpha_0^{(i)} + \alpha_1^{(i)}x_1 + \cdots + \alpha_{i-1}^{(i)}x_{i-1}\Big),$$

which has $\mathcal{O}(n^2)$ parameters.

NADE: Problem with FVSBN is that they only have a single hidden layer, making them not expressive. NADE uses MLPs:

$$\boldsymbol{h}_i = \sigma(\boldsymbol{W}_{:,1:i-1}\boldsymbol{x}_{1:i-1} + \boldsymbol{b}), \quad \hat{x}_i = \sigma(\boldsymbol{V}_{i,:}\boldsymbol{h}_i + c_i).$$

This model shares the parameters of $\boldsymbol{W}$ between timesteps, which means that it has $\mathcal{O}(nd)$ parameters.

MADE: Constructs an autoencoder which fulfills the autoregressive property. For this, we must ensure that there is no computational path between output unit $\hat{x}_{k+1}$ and any of $x_{k+1}, \dots, x_n$, relative to an arbitrary ordering. This is done by uniformly assigning integers 1 to n to each input unit and integer 1 to $n-1$ to each hidden unit. Then, we only allow values to propagate from units in layer ℓ to units in layer $\ell+1$ with equal or higher value. Finally, we allow connections between the last hidden layer and the output only to units with value that is strictly greater.

The problem with this approach is that it requires very large networks. And, while it is possible to train efficiently, sampling still requires n passes through the network.

Pixel-RNN: The idea is to generate image pixels starting from the corner and modeling the dependency on previous pixels using an RNN. This is slow, because of its sequential nature.

Pixel-CNN: We can solve the efficiency issue of Pixel-RNN by assuming that pixels only depend on a context region around them. This allows for parallelization during training. During training we need to make sure only previous pixels are used, thus we use a masked convolution. Blind spot $\Rightarrow$ Horizontal and vertical stacks of convolutions. To enforce the autoregressive property, we need to go over the color channels autoregressively.

WaveNet: Pixel-CNN (audio) with dilated convs for an exponential RF.

VRNN: RNNs are deterministic $\Rightarrow$ Add stoch. by sampling $\boldsymbol{h}_t$ from VAE: $\boldsymbol{z}_t \sim q_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{x}_t, \boldsymbol{h}_{t-1}), \boldsymbol{h}_t \sim p_{\boldsymbol{\theta}}(\cdot \mid \boldsymbol{h}_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t)$. Prior is learned.

Transformers: Stack normalization, MLPs, and self-attention:

$$\boldsymbol{Y} = \operatorname{softmax}\big(\boldsymbol{X}\boldsymbol{W}_Q\boldsymbol{W}_K^\top\boldsymbol{X}^\top/\sqrt{d} + \boldsymbol{M}\big)\boldsymbol{X}\boldsymbol{W}_V,$$

where $\boldsymbol{M}$ is a mask that masks out future timesteps with $-\infty$. Intuitively, the softmax computes how much attention should be given to certain values. Computational complexity is $\mathcal{O}(n^2d)$ with a maximum path length between input and output of $\mathcal{O}(1)$. This allows for easy error propagation during training.

Normalizing flow

$$\log|\det(\boldsymbol{A})^{-1}| = \log|\det(\boldsymbol{A})|^{-1} = -\log|\det(\boldsymbol{A})|$$
$$\det\big(\boldsymbol{I} + \boldsymbol{u}h'\boldsymbol{w}^\top\big) = 1 + h'\boldsymbol{u}^\top\boldsymbol{w}$$
$$\boldsymbol{x} \odot \boldsymbol{y} = \operatorname{diag}(\boldsymbol{x})\boldsymbol{y} = \operatorname{diag}(\boldsymbol{y})\boldsymbol{x}.$$

The determinant of a triangular matrix is the product of its diagonal.

Change of variables: Best of both worlds: latent space and a tractable likelihood by leveraging change of variables:

$$p_X(\boldsymbol{x}) = p_Z(f^{-1}(\boldsymbol{x}))|\det(\boldsymbol{J}_{\boldsymbol{x}}f^{-1}(\boldsymbol{x}))| = p_Z(\boldsymbol{z})|\det(\boldsymbol{J}_{\boldsymbol{z}}f(\boldsymbol{z}))|^{-1}.$$

Downside is that f must be invertible, which means that we must preserve dimensionality between latent space and data space. Furthermore the determinant of the Jacobian must be efficiently computed, thus we must design f such that its Jacobian is triangular.

Coupling layer:

$$f : \begin{bmatrix}\boldsymbol{x}_A \\ \boldsymbol{x}_B\end{bmatrix} \mapsto \begin{bmatrix}h(\boldsymbol{x}_A, \beta(\boldsymbol{x}_B)) \\ \boldsymbol{x}_B\end{bmatrix},$$

where β can be any NN and h is invertible w.r.t. its first argument, given the second. The inverse is:

$$f^{-1} : \begin{bmatrix}\boldsymbol{y}_A \\ \boldsymbol{y}_B\end{bmatrix} \mapsto \begin{bmatrix}h^{-1}(\boldsymbol{y}_A, \beta(\boldsymbol{y}_B)) \\ \boldsymbol{y}_B\end{bmatrix}.$$

Jacobian:

$$\boldsymbol{J}_{\boldsymbol{x}}f(\boldsymbol{x}) = \begin{bmatrix}\frac{\partial \boldsymbol{y}_A}{\partial \boldsymbol{x}_A} & \frac{\partial \boldsymbol{y}_A}{\partial \boldsymbol{x}_B} \\ \frac{\partial \boldsymbol{y}_B}{\partial \boldsymbol{x}_A} & \frac{\partial \boldsymbol{y}_B}{\partial \boldsymbol{x}_B}\end{bmatrix} = \begin{bmatrix}h'(\boldsymbol{x}_A, \beta(\boldsymbol{x}_B)) & h'(\boldsymbol{x}_A, \beta(\boldsymbol{x}_B))\beta'(\boldsymbol{x}_B) \\ \boldsymbol{0} & \boldsymbol{I}\end{bmatrix}.$$

To compute the det, we need the upper left and lower right matrices.

This layer leaves part of its input unchanged, thus we must make sure to alternate what parts of the input get transformed.

Composing transformations: Chaining many layers:

$$\boldsymbol{x} = f(\boldsymbol{z}) = (f_m \circ \cdots \circ f_1)(\boldsymbol{z}).$$

Using change of variables:

$$p_X(\boldsymbol{x}) = p_Z(f^{-1}(\boldsymbol{x}))\prod_{k=1}^{m}|\det(\boldsymbol{J}_{\boldsymbol{x}}f_k(\boldsymbol{x}))|^{-1}.$$

Training: Maximize the log-likelihood:

$$\log p_X(\boldsymbol{X}) = \sum_{i=1}^{n}\log p_Z(f^{-1}(\boldsymbol{x}_i)) + \sum_{k=1}^{m}\log|\det(\boldsymbol{J}_{\boldsymbol{x}}f_k(\boldsymbol{x}_i))|^{-1}.$$

NICE: Split data by partitioning into two subsets and randomly alternating which is given to the NN. Additive coupling network:

$$\begin{bmatrix}\boldsymbol{y}_A \\ \boldsymbol{y}_B\end{bmatrix} = \begin{bmatrix}\boldsymbol{x}_A + \beta(\boldsymbol{x}_B) \\ \boldsymbol{x}_B\end{bmatrix}.$$

RealNVP: Splits data by partitioning using a checkerboard and a channel-wise masking. The channel-wise masking is used after a squeezing operation to go from $C \times H \times W$ to $4C \times {}^{H}\!/_{2} \times {}^{W}\!/_{2}$. This ensures all data can interact with each other. Affine mapping:

$$\begin{bmatrix}\boldsymbol{y}_A \\ \boldsymbol{y}_B\end{bmatrix} = \begin{bmatrix}\boldsymbol{x}_A \odot \exp(s(\boldsymbol{x}_B)) + t(\boldsymbol{x}_B) \\ \boldsymbol{x}_B\end{bmatrix},$$

where s and t can be arbitrarily complex.

GLOW: Uses invertible 1×1 convolutions to split the data, meaning that it learns how to split. It consists of L levels, consisting of K steps of flow, which apply activation norm, invertible 1×1 convolution, and a coupling layer as in RealNVP, in that order.

Generative adversarial network

- KL div: $\mathrm{KL}(p\|q) = \mathbb{E}_{\boldsymbol{x}\sim p}[\log {}^{p(\boldsymbol{x})}\!/_{q(\boldsymbol{x})}] = -\mathbb{E}_{\boldsymbol{x}\sim p}[\log {}^{q(\boldsymbol{x})}\!/_{p(\boldsymbol{x})}]$.
- JS div: $\mathrm{JS}(p\|q) = {}^{1}\!/_{2}\mathrm{KL}(p\|{}^{(p+q)}\!/_{2}) + {}^{1}\!/_{2}\mathrm{KL}(q\|{}^{(p+q)}\!/_{2})$.

Problem with optimizing likelihood: Optimizing likelihood does not necessarily give good results. Two possible cases:

- Good likelihood with bad sample quality. Let p be a good model and q a model that only outputs noise. $0.01p + 0.99q$ has log-likelihood:
  $$\log(0.01p(\boldsymbol{x}) + 0.99q(\boldsymbol{x})) \geq \log(p(\boldsymbol{x})) - \log 100.$$
  The $\log p(\boldsymbol{x})$ is proportional to the dimensionality of the input. Thus, will be high for high-dimensional data.
- Low likelihood with high sample quality, which occurs when he model overfits on the training data. Results in bad likelihood on test set.

GAN: Solve the above problem by introducing a discriminator. The objective of the generator is then to maximize the discriminator's classification loss by generating images similar to the training set, implicitly inducing p_{model}. Value function:

$$V(D,G)=\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D(\boldsymbol{x})]+\mathbb{E}_{\boldsymbol{z}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I})}[\log(1-D(G(\boldsymbol{z})))].$$

Then, we have the following two-player zero-sum game:

$$\operatorname*{argmin}_G \operatorname*{argmax}_D V(D,G).$$

Optimal discriminator: $D^\star(\boldsymbol{x})=p_{\text{data}}(\boldsymbol{x})/p_{\text{data}}(\boldsymbol{x})+p_{\text{model}}(\boldsymbol{x})$.

Global optimality: The generator is optimal if $p_{\text{model}}=p_{\text{data}}$ and at optimum, we have

$$V(D^\star,G^\star)=-\log 4.$$

The generator implicitly optimizes the Jensen-Shannon divergence.

Convergence guarantee: Assuming that D and G have sufficient capacity, at each update step $D\to D^\star$, and p_{model} is updated to improve

$$V(D^\star,p_{\text{model}})=\mathbb{E}_{\boldsymbol{x}\sim p_d}[\log D^\star(\boldsymbol{x})]+\mathbb{E}_{\boldsymbol{x}\sim p_m}[\log(1-D^\star(\boldsymbol{x}))]$$
$$\propto \sup_D \int p_{\text{model}}(\boldsymbol{x})\log(1-D(\boldsymbol{x}))\mathrm{d}\boldsymbol{x}.$$

Then, p_{model} converges to p_{data}, because $V(D^\star,p_{\text{model}})$ is convex in p_{model} and supremum preserves convexity.

Weak result: G and D have finite capacity, D does not necessarily converge to $D^\star$, and due to the NN parametrization of G, the objective is no longer convex.

Generator loss saturates: Early in training, G is poor, which results in $\log(1-D(G(\boldsymbol{z})))$ saturating, i.e., going to $-\infty$. Instead, we should train G to maximize $\log D(G(\boldsymbol{z}))$.

Mode collapse: The generator learns to produce high-quality samples with low variability. Solution: Unrolled GAN, which optimizes the generator w.r.t. the last k discriminators.

Training instability: Optimizing two-player games lead to training instabilities, since making progress for one player may mean that the other player is worse off. Finding Nash-Equilibria is hard.

Optimizing JS divergence: If the supports of p_{model} and p_{data} are disjoint, it is always possible to find a perfect discriminator. This results in the loss function equaling zero, meaning that there will be no gradient to update the generator's parameters. The solution is use the Wasserstein GAN, which optimizes the Wasserstein distance. In this case, the loss does not fall to zero for disjoint supports, because it measures divergence by how different they are horizontally, rather than vertically. Intuitively, it measures how much "work" it takes to turn one distribution into the other.

Gradient penalty: To stabilize training, add a gradient penalty:

$$\mathbb{E}_{\boldsymbol{x}\sim p_d}\left[\log D(\boldsymbol{x})+\lambda\|\nabla D(\boldsymbol{x})\|^2\right]+\mathbb{E}_{\boldsymbol{x}\sim p_m}[\log(1-D(\boldsymbol{x}))].$$

Diffusion models

Diffusion models do not generate samples in single steps, but in many small steps. Starting from pure noise, they iteratively denoise it:

$$\boldsymbol{x}_0\sim p_d,\quad \boldsymbol{x}_T\sim\mathcal{N}(\mathbf{0},\boldsymbol{I}).$$

Diffusion: Governed by a noise schedule $\{\beta_t\}_{t=1}^T$:

$$q(\boldsymbol{x}_t\mid\boldsymbol{x}_{t-1})=\mathcal{N}(\sqrt{1-\beta_t}\boldsymbol{x}_{t-1},\beta_t\boldsymbol{I}).$$

Closed-form solution to efficiently compute training data:

$$\sqrt{\bar\alpha_t}\boldsymbol{x}_0+\sqrt{1-\bar\alpha_t}\boldsymbol{\epsilon},\quad \boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I}),$$

where $\alpha_t=1-\beta_t$ and $\bar\alpha_t=\prod_{i=1}^t\alpha_i$.

Denoising: Parametrize network to predict noise $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$. Iteratively denoise by

$$\boldsymbol{z}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I})\text{ if }t>1\text{, else }\boldsymbol{z}=\mathbf{0}$$
$$\boldsymbol{x}_{t-1}=\frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t-\frac{1-\alpha_t}{\sqrt{1-\bar\alpha_t}}\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)\right)+\sigma_t\boldsymbol{z}.$$

Training: Loss function:

$$\left\|\boldsymbol{e}-\boldsymbol{e}_{\boldsymbol{\theta}}\left(\sqrt{\bar\alpha_t}\boldsymbol{x}_0+\sqrt{1-\bar\alpha_t}\boldsymbol{\epsilon},t\right)\right\|^2,\quad \boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I}).$$

CLIP: Image-language model that has been trained on image-caption pairs. It consists of an image encoder and a text encoder network. By using a contrastive loss, CLIP is encouraged to encode the image and caption into similar embeddings.

Classifier guidance: Pretrain a classifier and guide the denoising in a direction favoring images that are more reliably classified by the classifier. This is done by injecting gradients of the classifier model into the sampling process. However, this requires training a very specific classifier on noisy data.

Classifier-free guidance: Jointly train a class-conditional and unconditional diffusion model. It then guides the generation process by

$$\boldsymbol{\epsilon}^\star(\boldsymbol{x},y;t)=(1+\rho)\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x},y;t)-\rho\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x};t)$$

In practice, we usually train a single model and just set the conditioning variable to all zero for the unconditional generation. Overall, guidance improves the quality, but reduces the diversity of outputs.

Latent diffusion models: Train VAE and perform diffusion on latent space for efficiency. The diffusion model then only needs to focus on the "semantic" aspect of generation.

Reinforcement learning

MDP: $(\mathcal{S},\mathcal{A},P,r,\gamma)$. **Policy**: $\pi:\mathcal{S}\to\Delta(\mathcal{A})$. At every point, the goal of the agent is to maximize:

$$G_t=\sum_{k=0}^\infty\gamma^k r(s_{t+k},a_{t+k})$$
$$=r(s_t,a_t)+\gamma G_{t+1}.$$

Value function and **Bellman equation**:

$$V^\pi(s)=\mathbb{E}_\pi[G_0\mid S_0=s]$$
$$=\sum_{a\in\mathcal{A}}\pi(a\mid s)\left[r(s,a)+\gamma\sum_{s'\in\mathcal{S}}P(s'\mid s,a)V^\pi(s')\right].$$

Bellman optimality operator:

$$(\mathcal{T}V)(s)\doteq\max_{a\in\mathcal{A}}\left\{r(s,a)+\gamma\sum_{s'\in\mathcal{S}}P(s'\mid s,a)V(s')\right\}.$$

This function is a γ-contraction and monotonic:

$$\max_{s\in\mathcal{S}}|(\mathcal{T}V')(s)-(\mathcal{T}V)(s)|\le\gamma\max_{s\in\mathcal{S}}|V'(s)-V(s)|$$
$$V(s)\le V'(s)\implies(\mathcal{T}V)(s)\le(\mathcal{T}V')(s).$$

The optimal value function $V^\star$ is the fixed-point of $\mathcal{T}$.

Value iteration: Iteratively apply $\mathcal{T}$. Linear convergence:

$$\max_{s\in\mathcal{S}}|V_t(s)-V^\star(s)|\le\gamma^t\max_{s\in\mathcal{S}}|V_0(s)-V^\star(s)|.$$

After convergence, we can get the optimal policy by acting greedily:

$$\pi^\star(s)\in\operatorname*{argmax}_{a\in\mathcal{A}}\left\{r(s,a)+\gamma\sum_{s'\in\mathcal{S}}P(s'\mid s,a)V^\star(s')\right\}.$$

Policy iteration: Alternates between computing the greedy policy and the value function of the policy.

Model-based and model-free: MB: Learn the underlying MDP and solve it. MF: Learn the policy or value function directly.

On-policy and off-policy: On-policy: Must learn from policy's own data. Off-policy: Can learn from any data.

Monte Carlo: On-policy method that learns from full trajectory and estimates the value function empirically:

$$V^\pi(s)\approx\frac{1}{n}\sum_{i=1}^n G(s)^i,$$

where $G(s)^i$ is the return of episode i, starting from s.

TD learning: Learn from transitions (s,a,r,s') using bootstrapping:

$$V(s)\leftarrow\alpha V(s)+(1-\alpha)(r+\gamma V(s')).$$

To find the optimal value function, we must visit all states sufficiently often. For this, we can use ϵ-greedy with Robbins-Monro conditions.

SARSA: Learns Q-values of a policy π (on-policy):

$$Q^\pi(s,a)\leftarrow\alpha Q^\pi(s,a)+(1-\alpha)(r+\gamma Q^\pi(s',a')),\quad a'\sim\pi(s').$$

Q-learning: Learns optimal Q-values (off-policy):

$$Q(s,a)\leftarrow(1-\alpha)Q(s,a)+\alpha(r+\gamma Q(s',a')),\quad a'\in\operatorname*{argmax}_{a\in\mathcal{A}}Q(s',a).$$

DQN: Large or infinite state spaces ⇒ Function approximation. DQN learns the Q-values for states. Loss function:

$$\ell(\boldsymbol{\theta})=(Q_{\boldsymbol{\theta}}(s,a)-(r+\gamma Q_{\bar{\boldsymbol{\theta}}}(s',a')))^2,\quad a'\in\operatorname*{argmax}_{a\in\mathcal{A}}Q_{\bar{\boldsymbol{\theta}}}(s',a).$$

We train as in supervised learning. Data is not i.i.d. ⇒ Replay buffer.

Sample inefficiency of deep RL: As the policy improves, we can collect better data. So, we have to keep training on new better data.

Policy search: Large or infinite action spaces ⇒ Parametrize $\pi_{\boldsymbol{\theta}}$:

$$\pi_{\boldsymbol{\theta}}(\cdot\mid s)=\mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}(s),\operatorname{diag}(\boldsymbol{\sigma}^2_{\boldsymbol{\theta}}(s))).$$

Probability of trajectory τ can be computed by

$$\pi_{\boldsymbol{\theta}}(\tau)=P(s_0)\prod_{t=0}^T\pi_{\boldsymbol{\theta}}(a_t\mid s_t)P(s_{t+1}\mid s_t,a_t).$$

Want trajectories with high return more likely $\Rightarrow$ Training objective:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right]$$
$$= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[r(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)],$$

using chain rule $\nabla \log f(\boldsymbol{x}) = \nabla f(\boldsymbol{x}) / f(\boldsymbol{x})$. We have

$$\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) = \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t).$$

Thus, the gradient does not depend on the MDP.

REINFORCE: The above is unbiased, but has high variance. We can reduce variance by introducing a baseline $b_t(\tau) = \sum_{t'=0}^{t-1} \gamma^{t'} r(s_{t'}, a_{t'})$, which turns $r(\tau)$ into G_t. This is on-policy.

Actor-critic: We can make it off-policy by estimating G_t by bootstrapping using a value network. This introduces bias, but reduces variance:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)(V(s_t, a_t) - (r(s_t, a_t) + \gamma V(s_{t+1}))).$$

Implicit surfaces and neural radiance fields

Voxels are bad because of $\mathcal{O}(n^3)$ memory cost. 3D points is bad because it does not model connectivity. Meshes are bad because of approximation error. We use learned implicit functions:

- *Occupancy network*: $f_{\boldsymbol{\theta}} : \mathbb{R}^3 \times \mathcal{Z} \to [0, 1]$ outputting probability of being inside mesh.
- *DeepSDF*: $f_{\boldsymbol{\theta}} : \mathbb{R}^3 \times \mathcal{Z} \to \mathbb{R}$ outputting distance to surface.

Implicit surfaces can do non-rigid transformations.

Training with watertight meshes: Simplest case. Sample n points that are either inside or outside the mesh. Train occupancy network by binary cross entropy. Train DeepSDF by regression of distance to mesh of the sampled points.

Training with point clouds: Train to make sure distance is 0 at points. However, this will result in trivial model that always outputs zero $\Rightarrow$ Eikonal term:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} |f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)|^2 + \lambda \mathbb{E}_{\boldsymbol{x}}\left[(\|\nabla_{\boldsymbol{x}} f_{\boldsymbol{\theta}}(\boldsymbol{x})\| - 1)^2\right].$$

This makes sense from a "distance" perspective, since we want to increase the distance by 1 when we move 1 unit away.

Training with images: Exponentially more data. High-level idea: Render model in same view as image and use photometric loss:

$$\ell(\hat{\mathbf{I}}, \mathbf{I}) = \sum_{\boldsymbol{u}} \|\hat{\mathbf{I}}_{\boldsymbol{u}} - \mathbf{I}_{\boldsymbol{u}}\|.$$

For this, we need a texture network $\boldsymbol{t}_{\boldsymbol{\theta}} : \mathbb{R}^3 \times \mathcal{Z} \to \mathbb{R}^3$ that outputs color. This requires the rendering pipeline to be differentiable.

Forward pass: For every pixel $\boldsymbol{u}$, determine the first intersection with surface $\hat{\boldsymbol{p}}$ using the secant method, which iteratively finds the linear intersection of the line connecting the points and the x-axis. $\hat{\mathbf{I}}_{\boldsymbol{u}} = \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})$.

Backward pass: Compute gradients (using implicit differentiation)

$$\frac{\partial \ell(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \sum_{\boldsymbol{u}} \frac{\partial \ell(\boldsymbol{\theta})}{\partial \hat{\mathbf{I}}_{\boldsymbol{u}}} \left(\frac{\partial^+ \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} - \frac{\partial \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \boldsymbol{w} \left(\frac{\partial f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \boldsymbol{w} \right)^{-1} \frac{\partial^+ f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} \right),$$

where $\boldsymbol{r}(d) = \boldsymbol{r}_0 + d\boldsymbol{w}$ is the ray connecting camera origin to $\boldsymbol{u}$.

NERF: Problem: Implicit surfaces are not good at complex scenes, especially with transparency or thin structures. NERF takes as input (x, y, z, θ, ϕ) and outputs (r, g, b, σ), where (θ, ϕ) are the view direction and σ is the density, allowing the modeling of transparency.

In the architecture, we must make sure that σ does not depend on (θ, ϕ). If σ depends on (θ, ϕ), it will overfit on the images and learn to remember images, rather than reconstruct the geometry. This is because the loss is based on the images.

Rendering: We render "volumetric clouds". Instead of only sampling at the surface, we sample along the whole ray and compute a weighted average. Let $\alpha_i = 1 - \exp(-\sigma_i \delta_i)$ with $\delta_i = t_{i+1} - t_i$ be the prob. of light stopping at point i. Then, we can compute the prob. of light reaching i by $T_i = \prod_{j=1}^{i-1} 1 - \alpha_j$. Then, we compute the final color:

$$\boldsymbol{c} = \sum_{i=1}^{n} T_i \alpha_i \boldsymbol{c}_i.$$

The fundamental difference with differentiable rendering is that we backpropagate through multiple points rather than a single point.

In general, NNs are biased toward low frequency functions, while we need high frequency functions. The solution is positional encodings. The low frequency function w.r.t. the encoding is then a high frequency function w.r.t. (x, y, z, θ, ϕ).

Gaussian splatting: Problem with NERF: Slow, because of the high number of samples. We can reduce by estimating a set of primitives around the object boundary and only sampling within these shapes. Cubes are hard to optimize. Spheres are bad at modeling thin structures. Solution: Model by many 3D Gaussians. The scene is initialized by a point cloud. Then, iteratively, we project the Gaussians onto the image plane and weight them similarly to NERF, compute the loss, and backpropagate to update the Gaussians. Note that there are no NNs. The weight of a Gaussian at a pixel $\boldsymbol{u}$ is computed by

$$\alpha_i(\boldsymbol{u}) = o_i \cdot \exp(-1/2 (\boldsymbol{x} - \boldsymbol{\mu}_i')^\top \boldsymbol{\Sigma}_i'^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_i')), \quad \boldsymbol{\Sigma}' = \boldsymbol{J}\boldsymbol{W}\boldsymbol{\Sigma}\boldsymbol{W}^\top \boldsymbol{J}^\top,$$

where o_i is the opacity, and $\boldsymbol{\mu}_i', \boldsymbol{\Sigma}'$ are the parameters of the 2D projection of the i-th 3D Gaussian. Simple covariance matrix:

$$\boldsymbol{\Sigma} = \boldsymbol{R}\boldsymbol{S}\boldsymbol{S}^\top \boldsymbol{R}^\top, \quad \boldsymbol{R} \in \mathbb{R}^{3 \times 3}, \boldsymbol{S} \in \mathbb{R}^3.$$

Parametric human body models

2D poses: Body modeling: Use the pictorial structure model, which models the body as a graph. Given an image $\mathbf{I}$ and vertex locations $L = [\ell_1, \ldots, \ell_k]$, we want to minimize score:

$$S(\mathbf{I}, L) = \sum_{i \in V} \alpha_i \cdot \phi(\mathbf{I}, \ell_i) + \sum_{i,j \in E} \beta_{ij} \psi(\ell_i, \ell_j).$$

Generalize by assigning mixture components m_i:

$$S(\mathbf{I}, L, M) = \sum_{i \in V} \alpha_i^{m_i} \phi(\mathbf{I}, \ell_i) + \sum_{i,j \in E} \beta_{ij}^{m_i m_j} \psi(\ell_i, \ell_j) + \sum_{i,j \in E} b_{ij}^{m_i m_j}.$$

where $\alpha_i^{m_i}$ is the "local appearance template" for part i with type m_i, $\beta_{ij}^{m_i m_j}$ expresses the likelihood of having template m_i for part i and template m_j for part j given the distance between ℓ_i and ℓ_j, and $b_{ij}^{m_i m_j}$ is the pairwise co-occurrence prior.

Feature learning: Use deep learning to either regress the locations (DeepPose) or output heatmaps for each vertex (Convolutional Pose Machine). They both use architectures with a refinement process.

The two can also be combined by first using feature learning and then refining with body modeling.

Linear-blend skinning: Simplest method that transforms vertices as a weighted linear combination of global joint transformations:

$$\bar{\boldsymbol{t}}_i' = \left(\sum_k w_{ki} \boldsymbol{G}_k'(\boldsymbol{\theta}, \boldsymbol{J}) \right) \bar{\boldsymbol{t}}_i, \quad \boldsymbol{G}_k'(\boldsymbol{\theta}, \boldsymbol{J}) = \boldsymbol{G}_k(\boldsymbol{\theta}, \boldsymbol{J}) \boldsymbol{G}_k(\boldsymbol{\theta}^\star, \boldsymbol{J})^{-1},$$

where k are bones and i are vertices, $\boldsymbol{G}_k(\boldsymbol{\theta})$ is the rigid bone transformation for bone k, which is weighted by w_{ki} which determines the influence of bone k on vertex i.

Skinned multi-person linear model: Problem with LBS: Does not account for variation in body shape and poses often result in unwanted deformations. SMPL solves this by encoding a body shape parameter $\boldsymbol{\beta} \in \mathbb{R}^{10}$ and pose parameter $\boldsymbol{\theta} \in \mathbb{R}^{9K}$.

1. Translate template to identity mesh: $\bar{\boldsymbol{T}} + \boldsymbol{B}_S(\boldsymbol{\beta})$.
2. Correct for future deformations: $\bar{\boldsymbol{T}} + \boldsymbol{B}_S(\boldsymbol{\beta}) + \boldsymbol{B}_P(\boldsymbol{\theta})$.
3. Linear-blend skinning:

$$\bar{\boldsymbol{t}}_i' = \left(\sum_k w_{ki} \boldsymbol{G}_k'(\boldsymbol{\theta}, \boldsymbol{J}(\boldsymbol{\beta})) \right) (\bar{\boldsymbol{t}}_i + \boldsymbol{b}_{S,i}(\boldsymbol{\beta}) + \boldsymbol{b}_{P,i}(\boldsymbol{\theta})).$$

We learn $\boldsymbol{B}_S(\boldsymbol{\beta})$ by PCA on a body shape dataset. $\boldsymbol{\beta}$ are the linear weights of the largest principal components:

$$\boldsymbol{B}_S(\boldsymbol{\beta}; \mathcal{S}) = \sum_{n=1}^{|\boldsymbol{\beta}|} \beta_n \boldsymbol{S}_n.$$

We learn $\boldsymbol{B}_P(\boldsymbol{\theta})$ from a body pose dataset:

$$\boldsymbol{B}_P(\boldsymbol{\theta}; \mathcal{P}) = \sum_{n=1}^{9K} (\boldsymbol{R}_n(\boldsymbol{\theta}) - \boldsymbol{R}_n(\boldsymbol{\theta}^\star)) \boldsymbol{P}_n,$$

where $\boldsymbol{\theta}^\star$ is the template pose, $\boldsymbol{R}(\boldsymbol{\theta})$ maps pose vectors to vectors of rotation matrices, and $\boldsymbol{P}_n \in \mathbb{R}^{3N}$ is a vector of vertex displacements

Learned gradient descent: Method for fitting 3D human shapes to images by combining gradient-based optimization with NNs. It leverages a NN to predict the parameter update rule for each optimization iteration.

$$\mathcal{L}(\Theta_n) = L_{\text{reproj}}(\hat{\boldsymbol{x}}_n, \boldsymbol{x}_{\text{GT}})$$
$$\Delta = \mathcal{N}_{\boldsymbol{w}}\left(\frac{\partial \mathcal{L}(\Theta_n)}{\partial \Theta_n}, \Theta_n, \boldsymbol{x}_{\text{GT}} \right)$$
$$\Theta_{n+1} = \Theta_n + \Delta,$$

where $\mathcal{N}_{\boldsymbol{w}}$ is a NN that predicts the update.