Deep learning optimization

Pierre Hellier

Content

- Introduction to deep networks
- The computational graph and automatic differentiation (forward pass, backpropagation)
- Neural network tricks:
  - dropout
  - initialization
  - GD with momentum
  - Learning rate scheduler

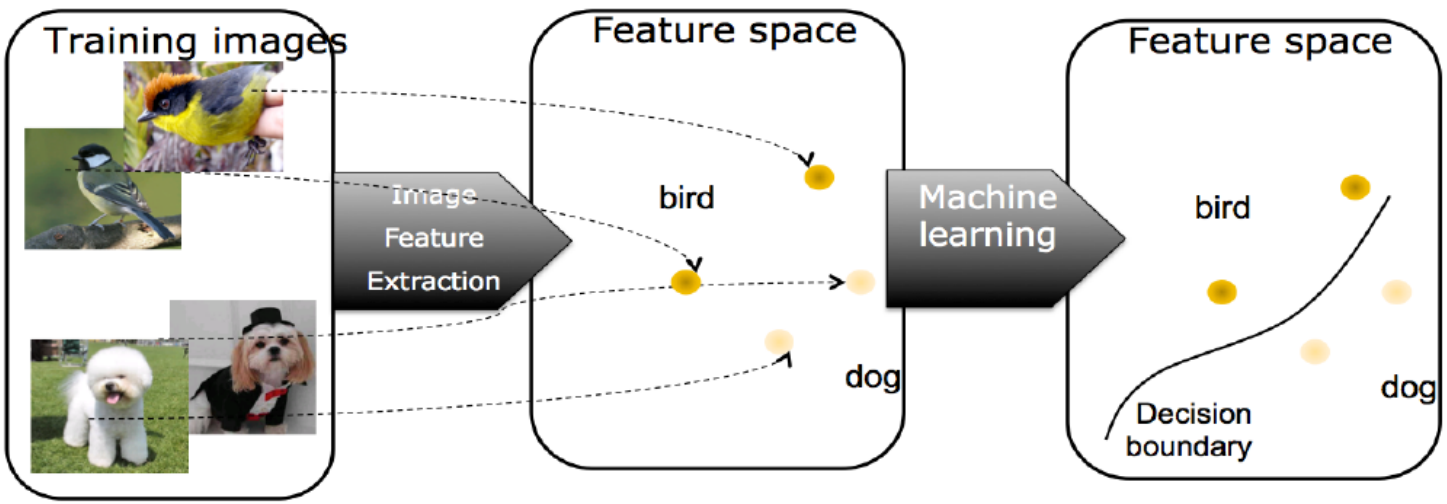
Deep learning introduction

History of Deep learning

- Deep learning is a branch of machine learning
- Originally discovered in the 50's
- Gone into shadows for some decades
- Revival since enormous successes in 2006 (speech) and 2012 (image)
- Learn universal, hierarchical, compositionnal functions

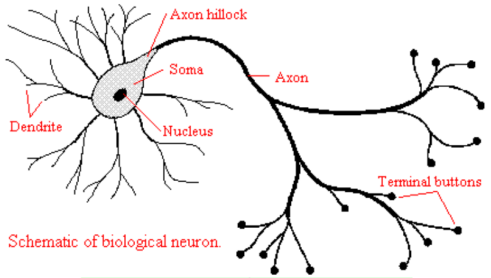
Before Deep learning

- Measured signal: image/audio
- Handcrafted representation features (Gabor/fourier, sift, mfcc, *etc.*)
- Classification in the feature space

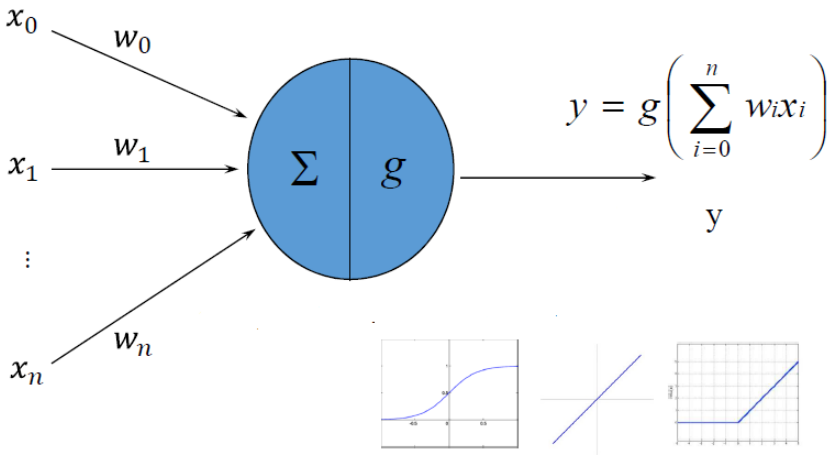


The neuron model

- First proposed by Rosenblatt, 1957



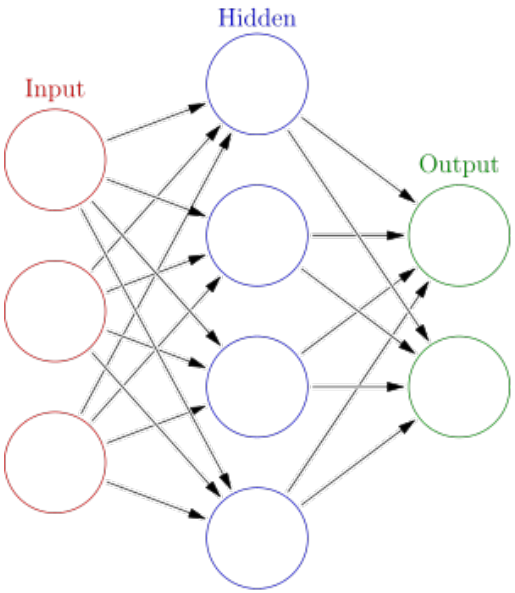
- loosely bio-inspired
- Weighted sum of input, followed by non-linear activation

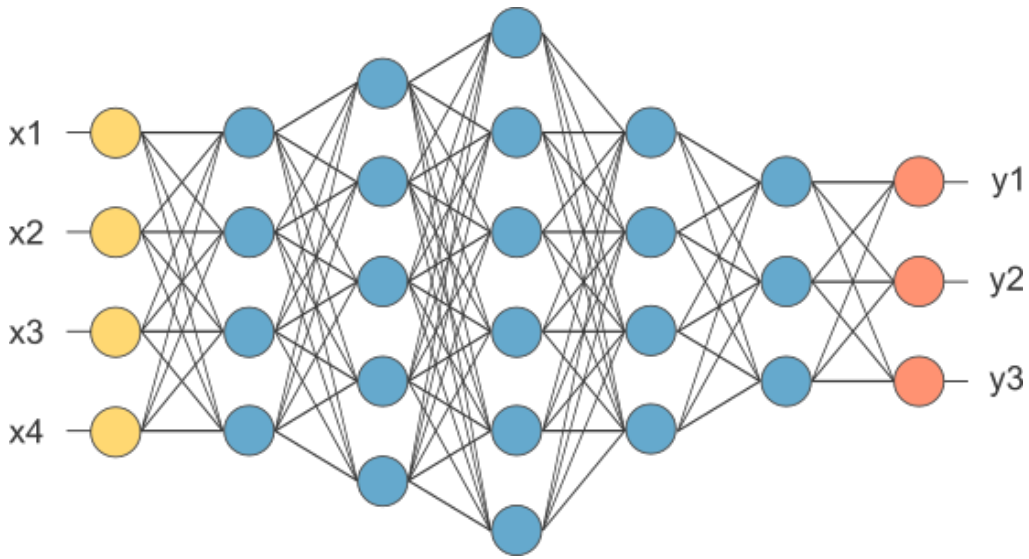


Bio-inspired (and not copied)

Neural network

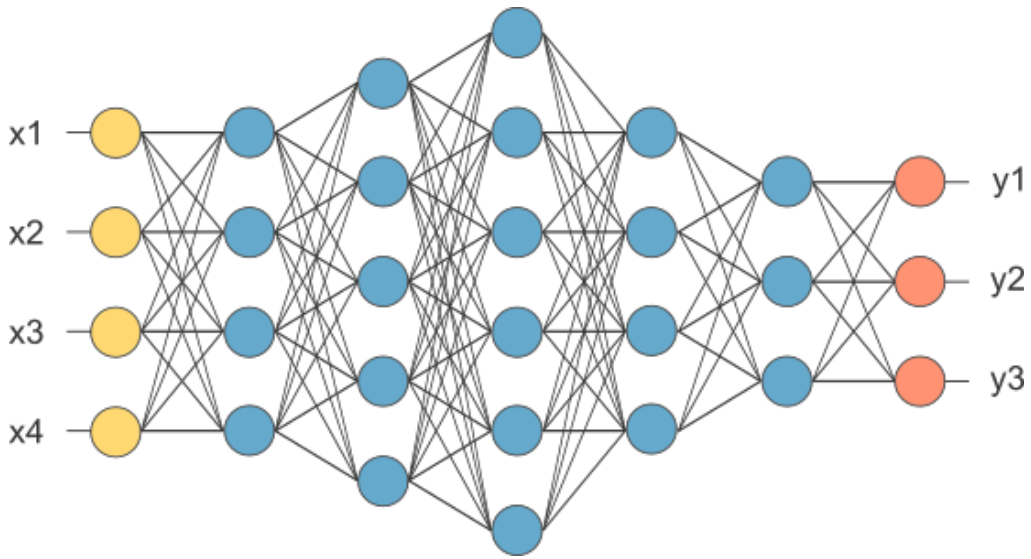
- A neural network (NN) is a composition of neural layers
- From shallow networks (1 layer) to deep networks (10-100 layers)





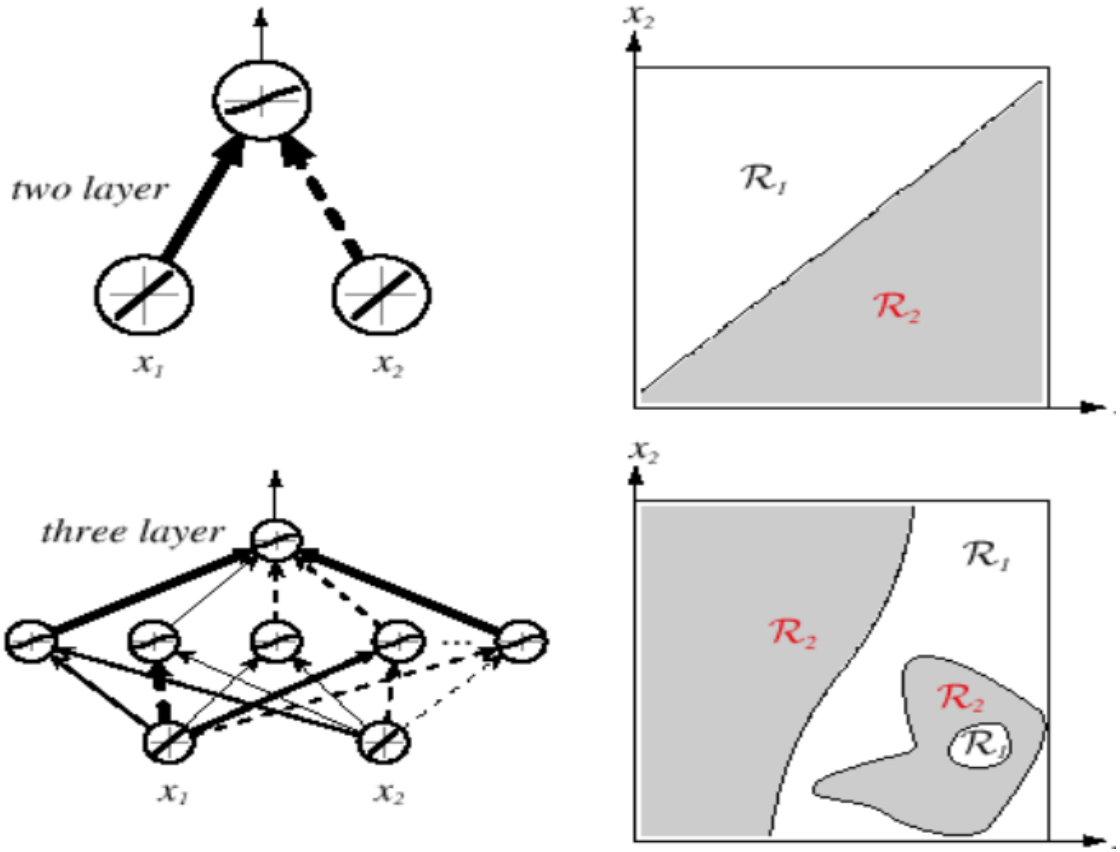
Neural network

- A neural network leads to a composition of function
- Layer l can be written in matrix form
  $X_l = f_l(X_{l-1}) = g(W_l X_{l-1})$
- For k layers, output $Y = f_k \circ f_{k-1} \circ \cdots \circ f_2 \circ f_1(X)$



Why deeper?

Depth leads to a more complex function learning



Richard *et al.*, Pattern Classification, 2001

Universal function approximation

Feedforward networks are universal function approximation:

$$\forall f \in \mathcal{C}([0,1]^n), \forall \epsilon > 0; \exists F \text{ s.t. } \forall x \in [0,1]^n, |F(x) - f(x)| < \epsilon)$$

With $F(x) = \sum_i v_i \phi(w_i^T x + b_i)$, where ϕ is non-constant, bounded, monotonically-increasing continuous function[1]

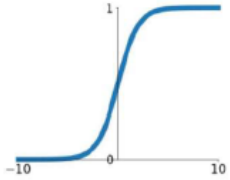
[1] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. Neural networks, 4(2), 251-257.

Activation functions

- Activation functions are necessary to introduce non-linearities
- Classically, sigmoid function: issue of vanishing gradient, problematic for gradient descent and optimization.
- RELU is the most used one and leads to networks that can be more efficiently trained

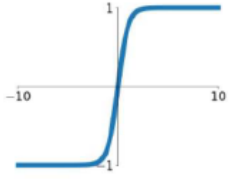
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



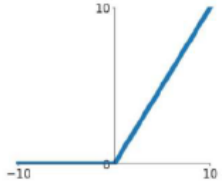
tanh

$$\tanh(x)$$



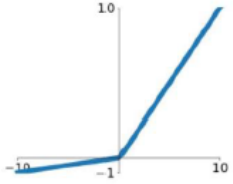
ReLU

$$\max(0, x)$$



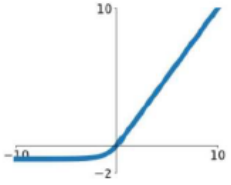
Leaky ReLU

$$\max(0.1x, x)$$



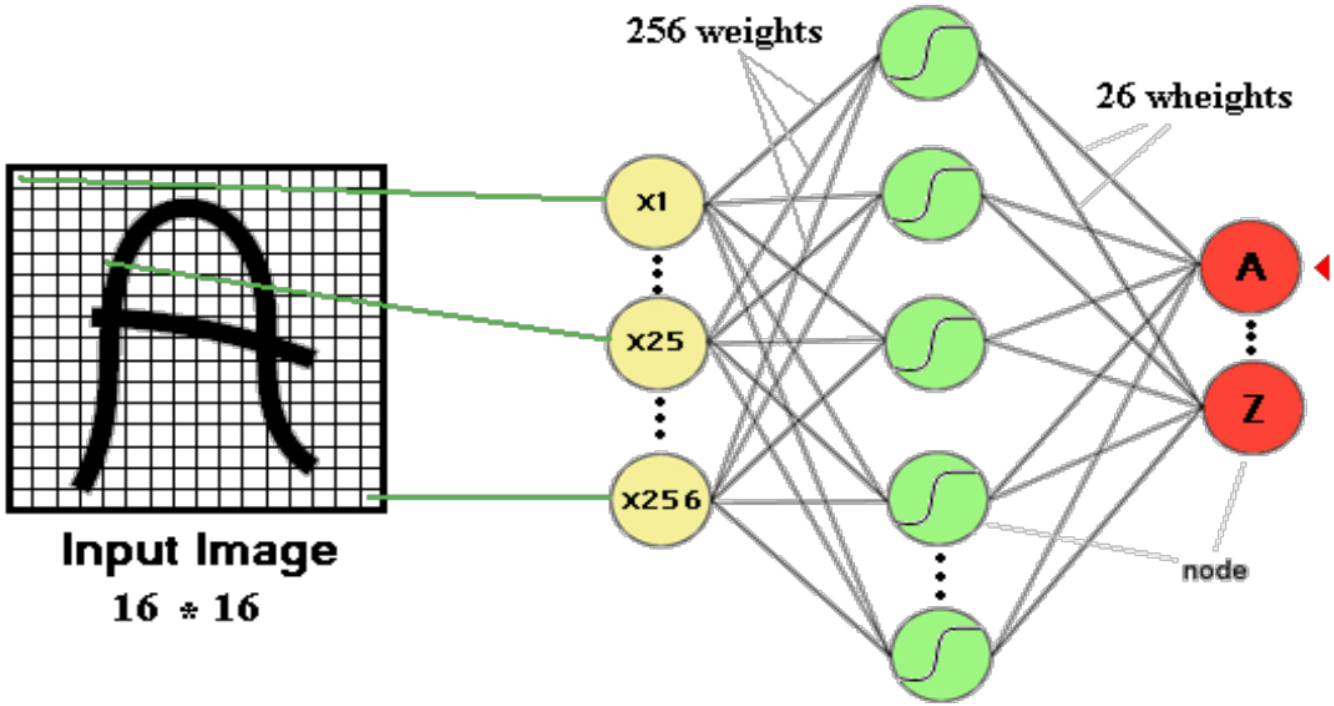
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Neural nets and images

- Images (and other signals) have a structure
- Brut force connection
  - Rasterize the signal
  - Connect each pixel as an input



Neural nets and images

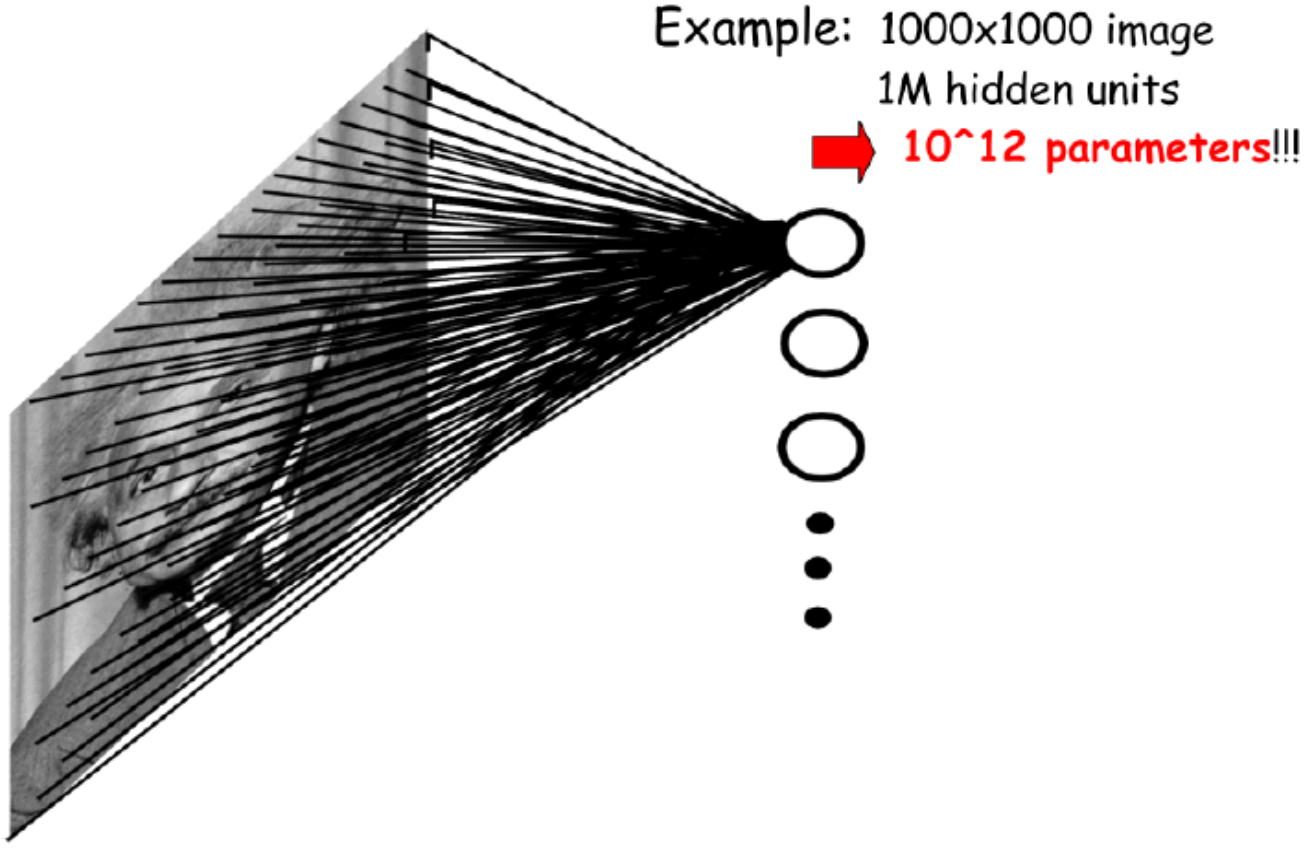
Two main issues

- Scalability
- Stability

Convolutional networks solve these two issues!

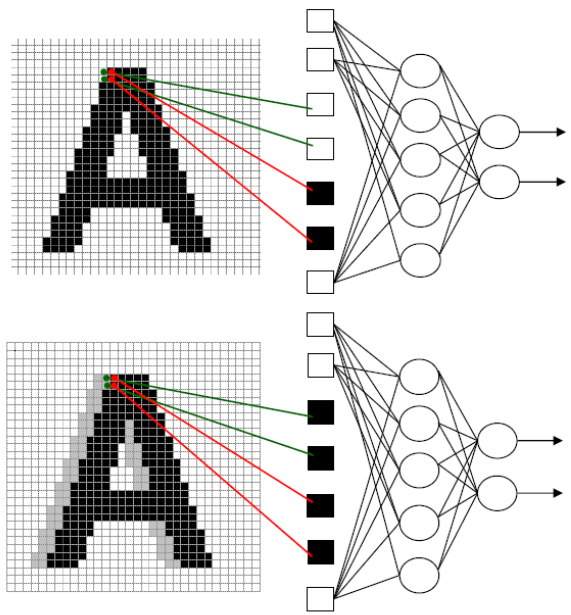
Scalability issue

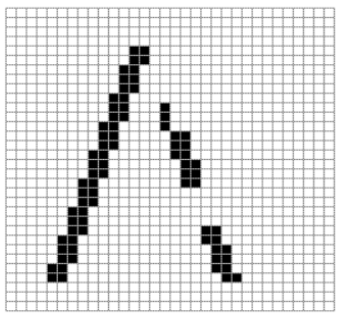
- For a 1000.1000 image, 10^6 pixels
- For the first fully connected layer, 10^{12} neurons!



Stability issue

- A small change of the image shall lead to a small change of the representation
- An image, slightly translated, shall be recognized as the same image



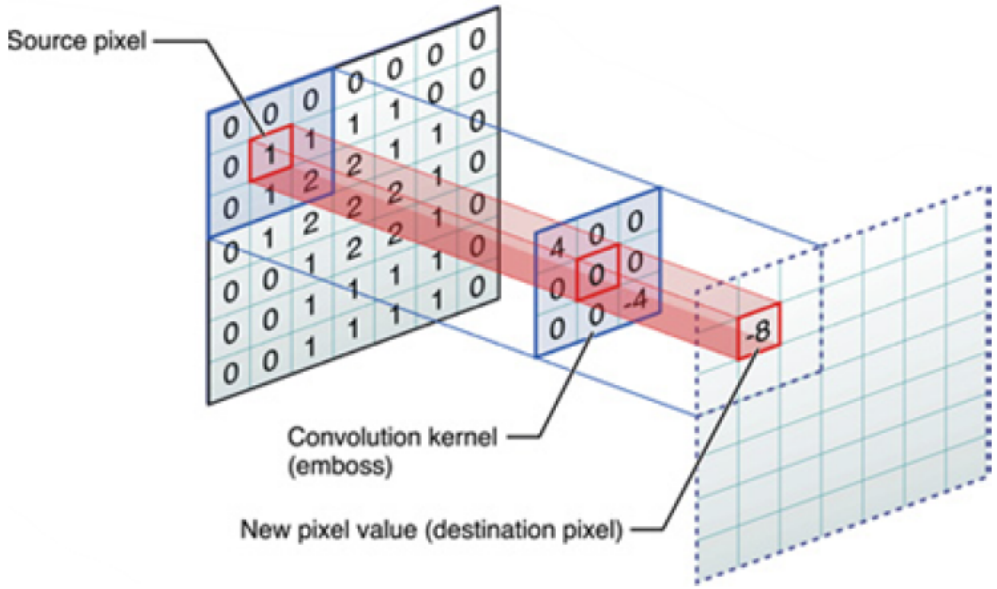


154 input change
from 2 shift left
77 : black to white
77 : white to black

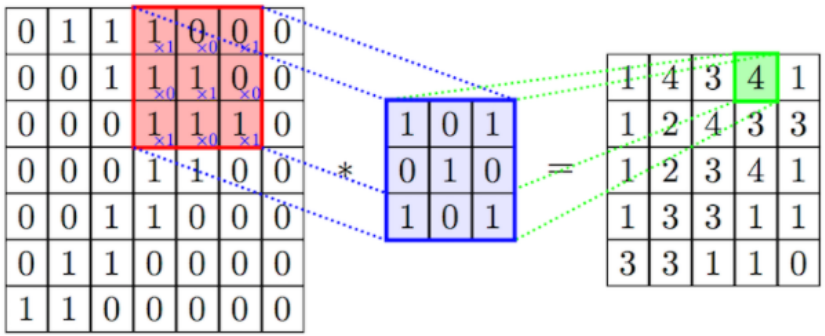
Credit: Yann LeCun

Convolutionnal networks

- Some signals like images have local stationnarities to be exploited
- A convolution captures information and is more compact
- $f * g = \int_t f(t)g(x-t)dt$
- ConvNets are used in all computer vision tasks



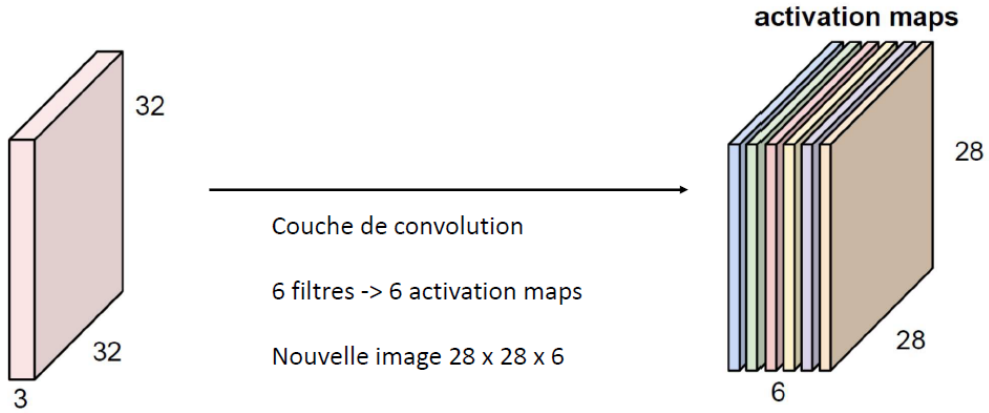
Convolutionnal networks



Convolutionnal networks

The convolution kernel slides on the entire image

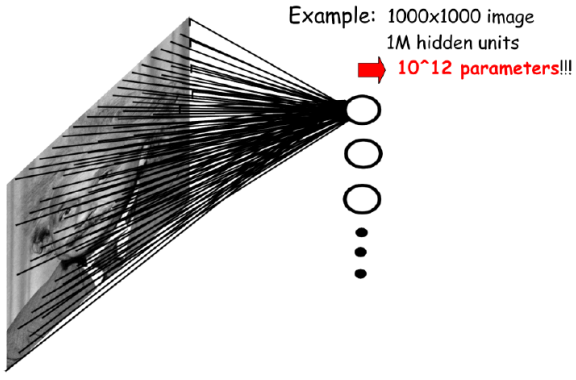
- Weights are shared: solve scalability issue
- Each convolution leads to an activation map

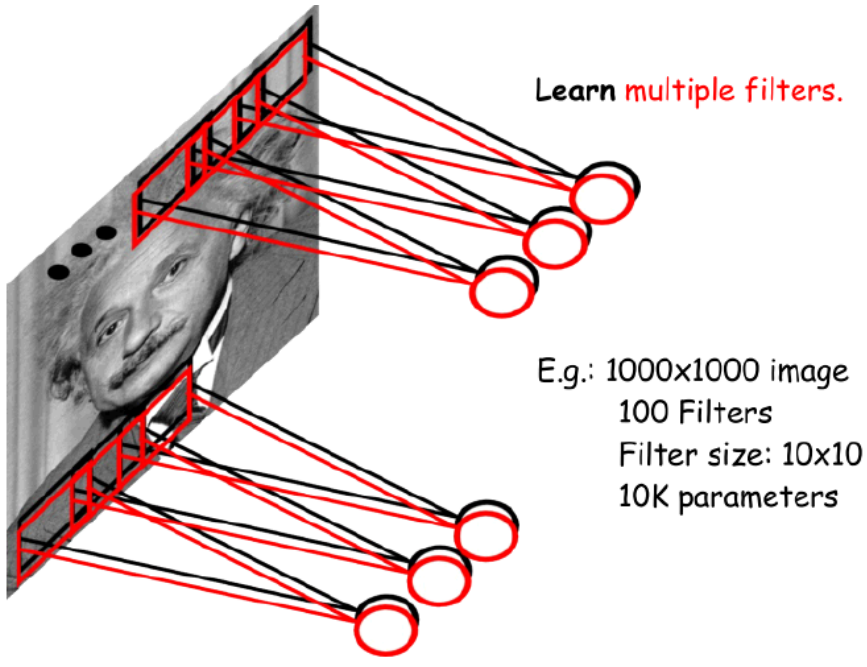


Convolutionnal networks

The convolution kernel slides on the entire image

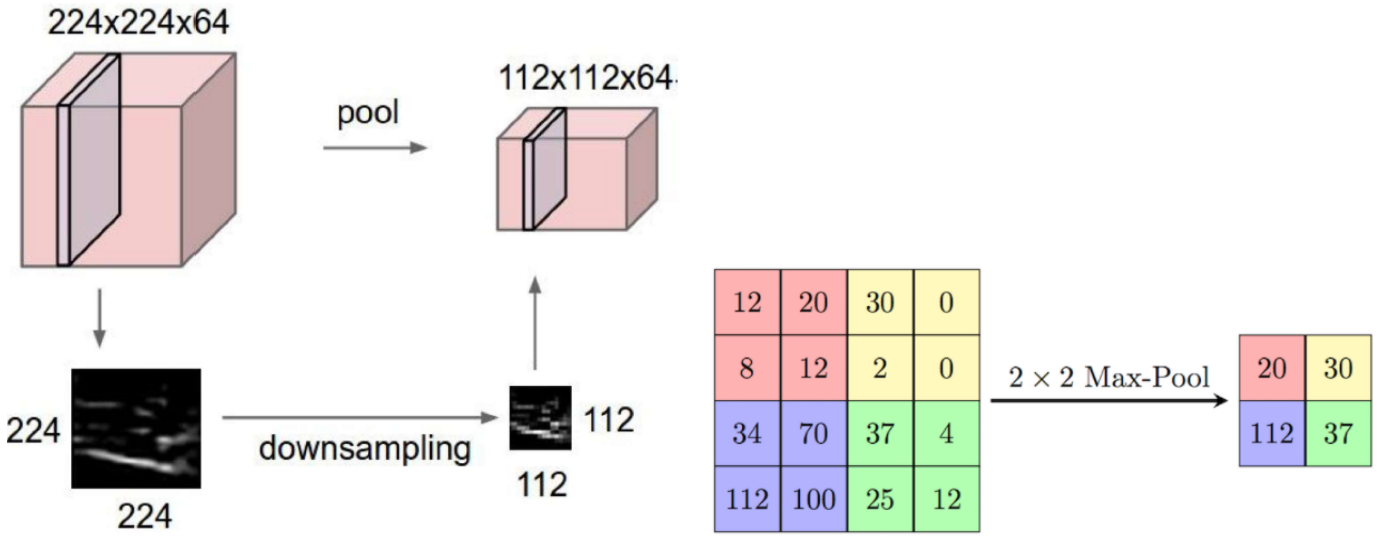
- Weights are shared: solve scalability issue
- Each convolution leads to an activation map



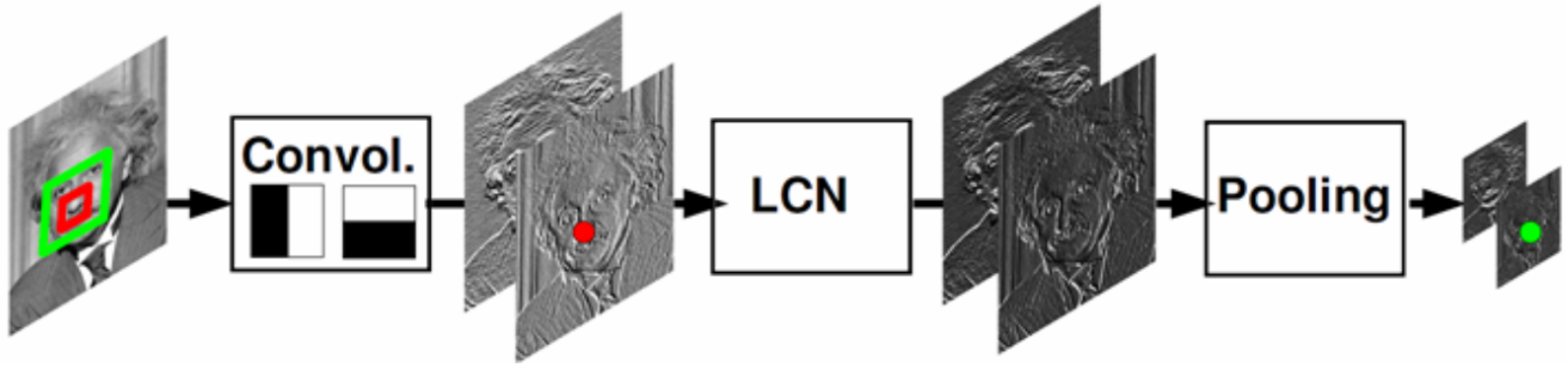


Pooling

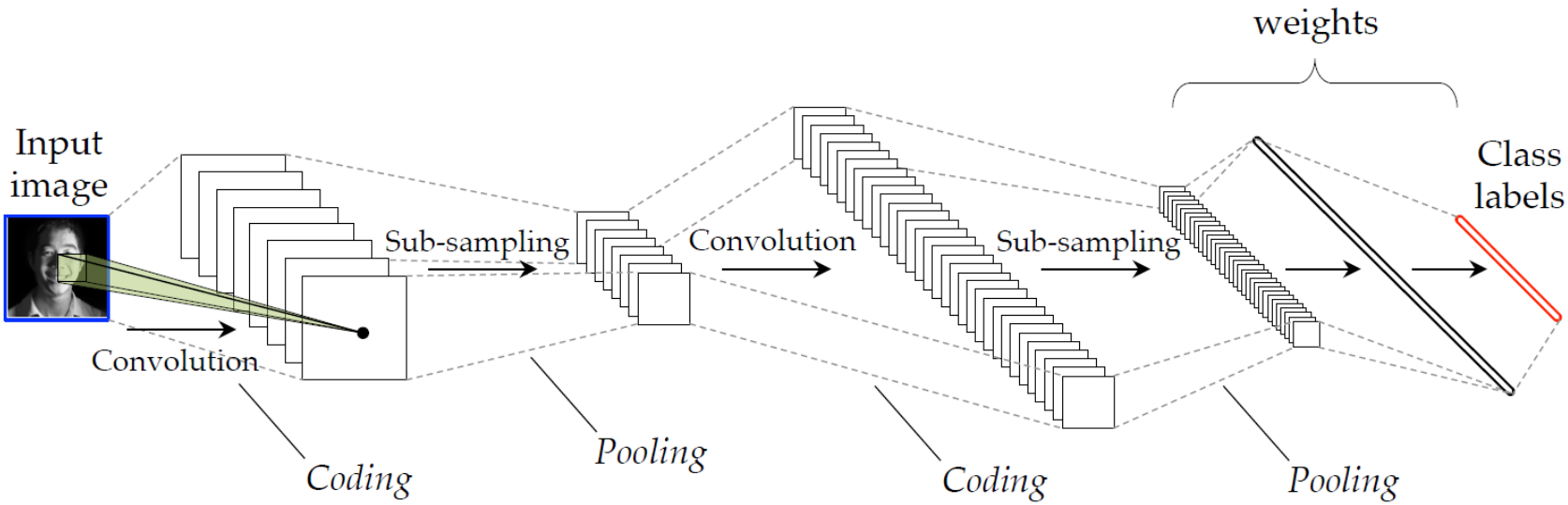
Pooling (subsampling) is used to reduce dimension after each convolution: increases invariance and solves the stability issue



Pooling leads to (some) invariance



Typical CNN


weights
Input image
Sub-sampling
Convolution
Sub-sampling
Class labels
Convolution
Coding
Pooling
Coding
Pooling

Link with scale space theory

- Concept: from a given image, construct a pyramid of convoluted images (Gaussian kernel) and increase scale. Witkin 1983, Koenderink 1984.
- Somehow related to the human vision system (receptive field profiles in the mammalian retina and visual cortex can be well modeled by linear Gaussian derivative operators)
- Applied to feature detection, feature classification, image segmentation, image matching, motion estimation, computation of shape cues and object recognition
- Classification: relevant details of images exist only over a restricted range of scale.

Scale space

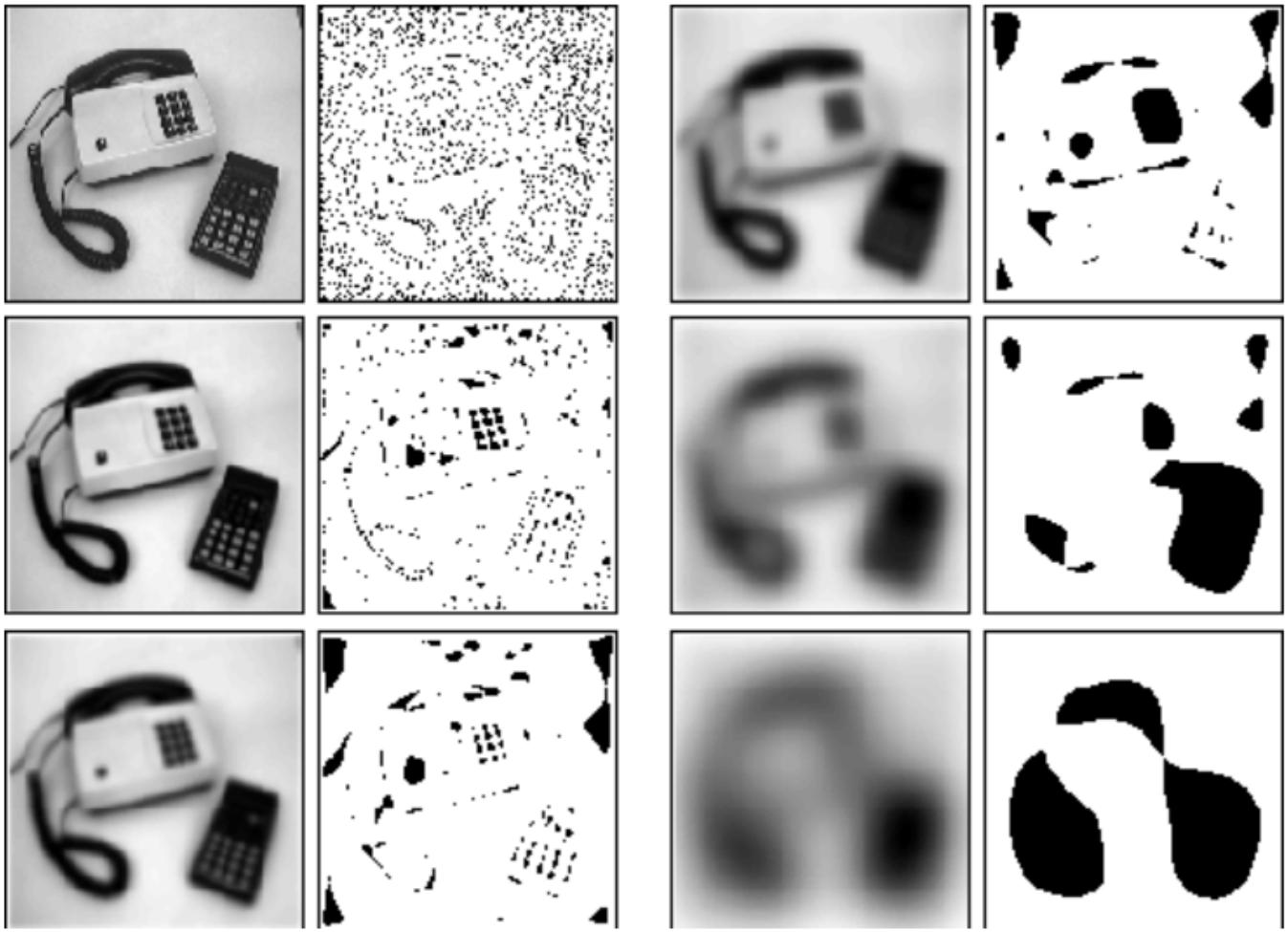
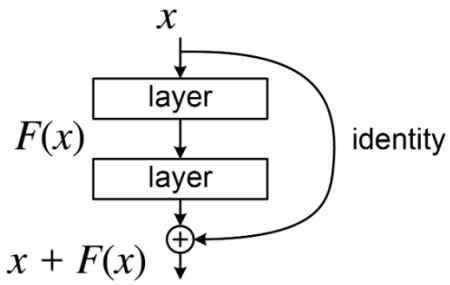


Image across scales and blobs of local image minima

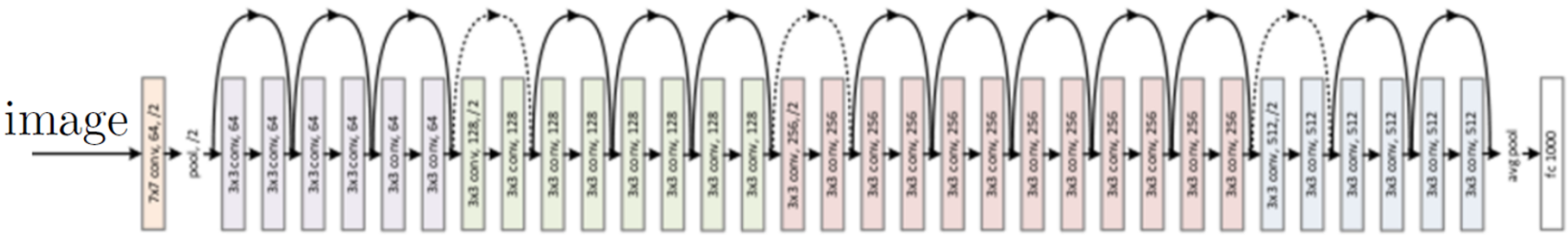
Residual networks

Resnets

He, Kaiming, et al. "Deep residual learning for image recognition." *CVPR* 2016.

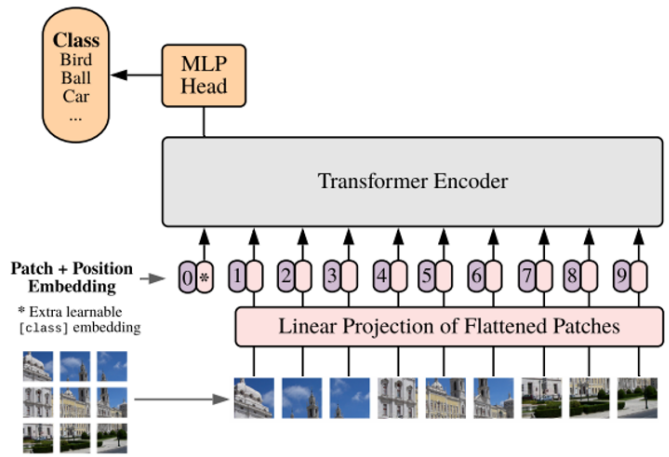


- Introduce skip connections in network
- Facilitates back-propagation in very deep networks
- Backprop = automatic backward differentiation for gradient computation
- Vanishing gradient = edge of the computational graph leads to (almost) zero gradient

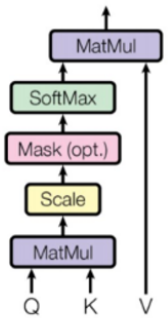


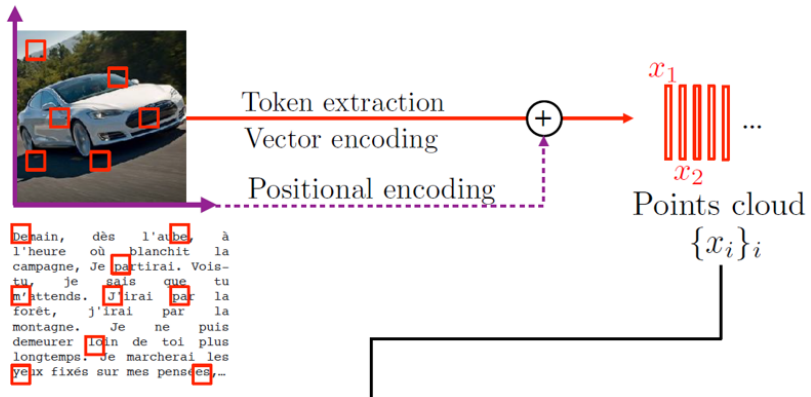
Vision Transformers (1)

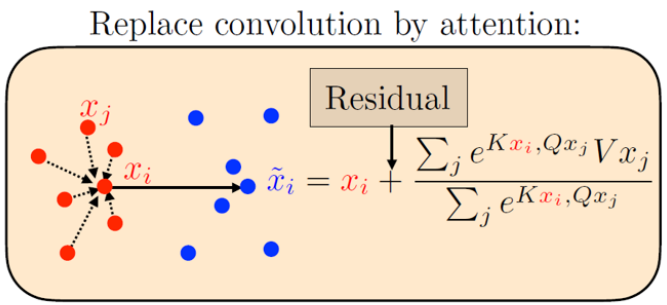
- Transformer applied to CV
- Process:
  - Patch extraction
  - Linear embedding + position embedding
  - Fed into transformer
- Now SOTA for (almost) all vision tasks
- Evolutions:
  - Masked autoencoder with two VIT
  - DINO (self-**di**stillation with **no** labels), self supervision
  - Swin transformers

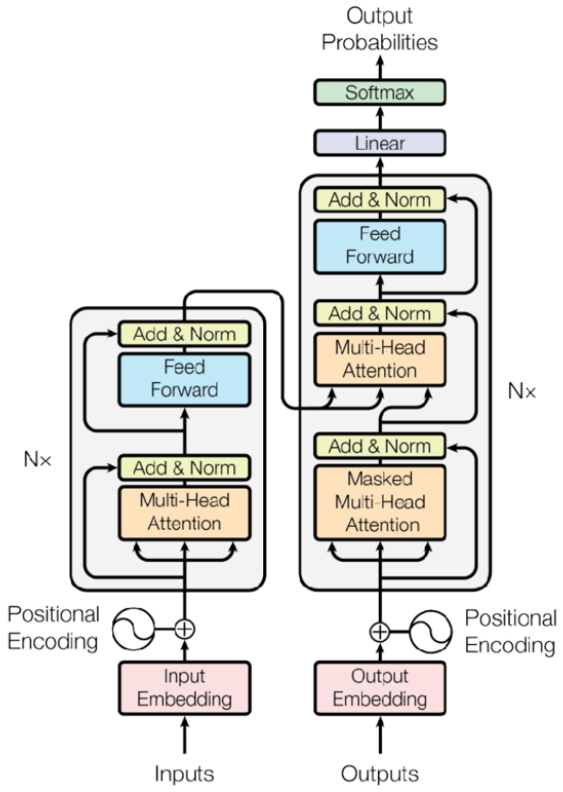


Vision Transformers (2)









Gabriel Peyre (https://www.gpeyre.com/)

Vision Transformers (3)

Transformers are universal context learners

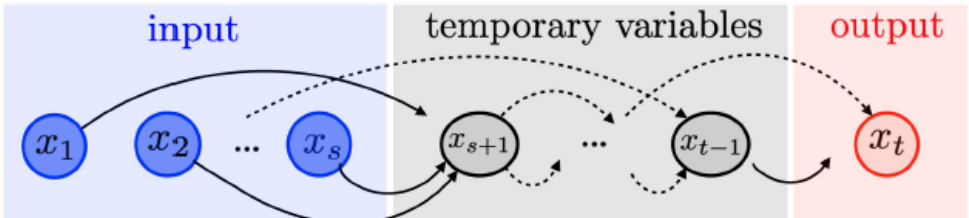
Furuya, Takashi, Maarten V. de Hoop, and Gabriel Peyré. "Transformers are universal in-context learners." *arXiv preprint arXiv:2408.01367* (2024).

- Extension of MLP as universal functions approximator [Cybenko 89; Hornik 89]
- In-context = map tokens to tokens, and map depends on previously seen tokens
- Parameters $\Theta \equiv \{K, Q, V\}$ for each attention head
- Multi-attention is always used
- #tokens is generally large (large context)
- Universal in-context learner, for fixed precision:
  - Single transformer can operate on arbitrary #tockens
  - Fixed embedding dimension (not related to precision)
  - Fixed number of heads (proportional to dimension)
- Does not provide the full receipe, but some useful insights

Computational graph and automatic differentiation

Computational graph

- Let us consider an output function f, depending on input variables $x_1, \ldots, x_s$, with intermediate computed variables $x_{s+1}, \ldots, x_t$, hence $f(x = \{x_1, \ldots, x_s\}) = x_t$
- Gradient computation through finite differences $\nabla_i f(x) = \frac{1}{\epsilon}(f(x + \epsilon\delta_i) - f(x))$, requires s evaluation of the function
- For NN, the number of parameters is extremely large, hence the numerical approximation of gradient is unfeasible in practice
- Solution: automatic differentiation (AD) on graph
- Algorithm can be represented with directed acyclic graph, linking output variable to input
- Evaluation of function = forward traversal of graph



Forward mode of automatic differentiation

- Forward mode: compute all derivatives of internal variables $x_k, k \in \{s+1, \ldots, t\}$ w.r.t. the input, hence

$$\forall i \in \{1, \ldots, s\}, \forall k \in k \in \{s+1, \ldots, t\}, \frac{\partial x_k}{x_i} = \sum_{l \in \mathcal{P}(k)} \frac{\partial x_k}{x_l} \frac{\partial x_l}{x_i}$$

- Where $\mathcal{P}(k)$ denotes all nodes $l < k$ that are connected to node k

Backward mode of automatic differentiation

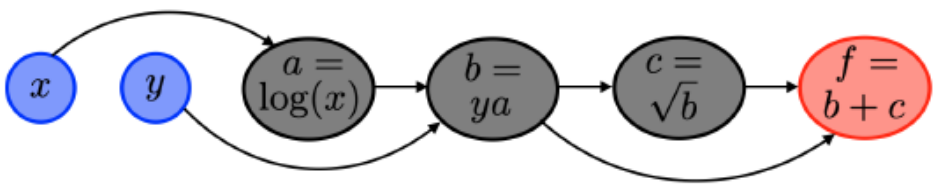
- Backward mode: iterative process from the end graph node x_t
- Recursively compute derivatives of x_t w.r.t. internal variables $x_k, k \in \{s+1, \ldots, t\}$, hence

$$\forall k \in k \in \{s+1, \ldots, t\}, \frac{\partial x_t}{x_k} = \sum_{m \in \mathcal{S}(k)} \frac{\partial x_t}{x_m} \frac{\partial x_m}{x_k}$$

- Where $\mathcal{S}(k)$ denotes all *son* nodes of node k
- Enables to compute all derivatives in one pass: more efficient than forward mode

Automatic differentiation example

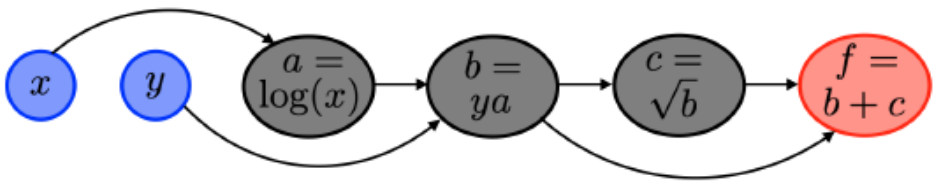
- Let us consider the function $f(x, y) = y \log(x) + \sqrt{y \log(x)}$
- Here the graph is easy to draw
- Note: in general, the user must provide the graph (explicit in neural networks)



Exercice of automatic differentiation

Problem

- Compute the forward mode of automatic differentiation
- Compute the backward mode of automatic differentiation



Solution for forward gradient computation

- $\frac{\partial a}{\partial x} = \frac{1}{x}$
- $\frac{\partial b}{\partial x} = \frac{\partial b}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial b}{\partial y}\frac{\partial y}{\partial x} = y\frac{1}{x}$
- $\frac{\partial c}{\partial x} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial x} = \frac{1}{2\sqrt{b}}\frac{y}{x}$
- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial x} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial x} = \frac{y}{x} + \frac{1}{2\sqrt{b}}\frac{y}{x}$

And we need another pass to compute $\frac{\partial f}{\partial y}$!

Solution for backward gradient computation

- $\frac{\partial f}{\partial c} = 1$
- $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial b} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} = 1 + \frac{1}{2\sqrt{b}}$
- $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} = (1 + \frac{1}{2\sqrt{b}})y$
- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} = \frac{1}{x}(1 + \frac{1}{2\sqrt{b}})y$
- $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial y} + \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} = (1 + \frac{1}{2\sqrt{b}})a$

In one pass, we can compute both gradients!

Special case of function composition

- The simplest case for the computational graph is the composition of functions, i.e., $f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$, with $f_k : \mathbb{R}^{n_{k-1}} \rightarrow \mathbb{R}^{n_k}$
- At each step k, the gradient matrix is the Jacobian matrix of partial derivatives $A_k = \partial f_k(x_{k-1}) \in \mathbb{R}^{n_k \times n_{k-1}}$
- The full Jacobian is then $\partial f(x) = A_t \times A_{t-1} \times \cdots \times A_2 \times A_1$

Special case of function composition: complexity

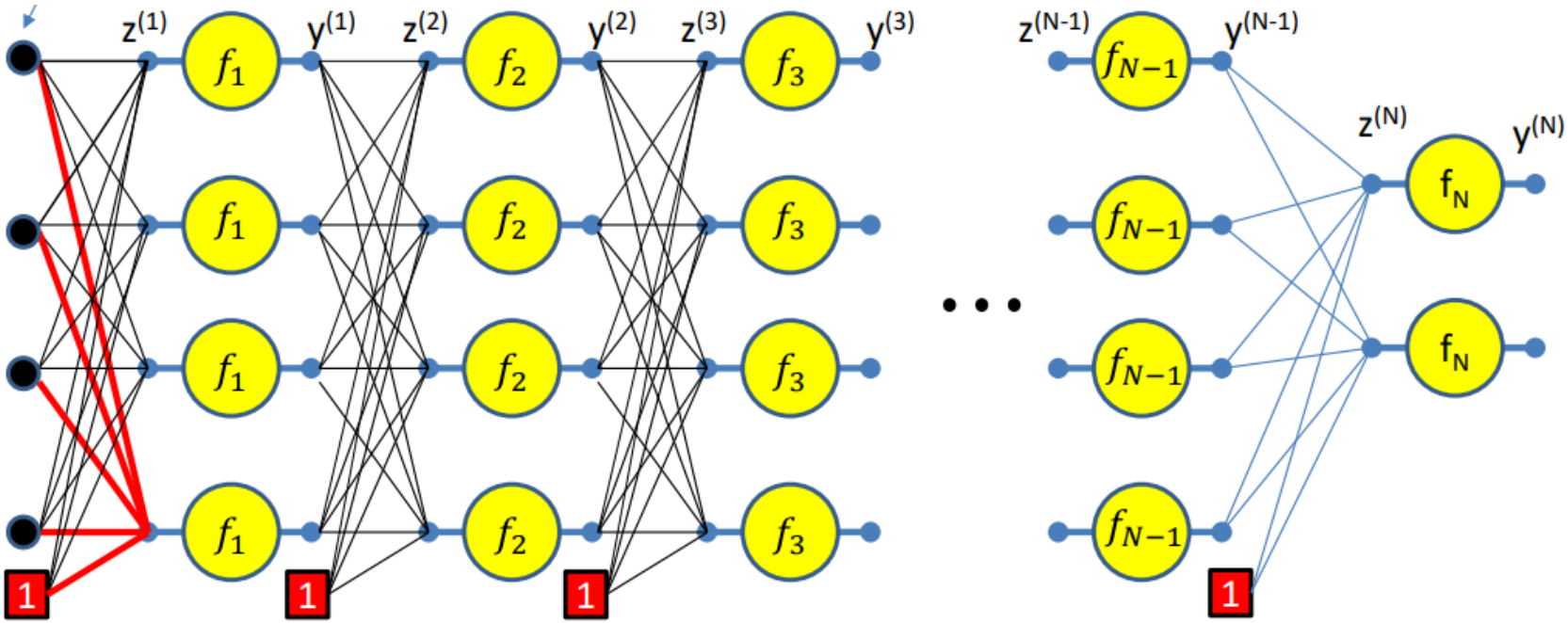
- The full Jacobian is then $\partial f(x) = A_t \times A_{t-1} \times \cdots \times A_2 \times A_1$
- Operation order for forward mode:
  $\partial f(x) = A_t \times (A_{t-1} \times (\cdots \times (A_2 \times A_1)))$
- Operation order for backward mode:
  $\partial f(x) = (((A_t \times A_{t-1}) \times \cdots \times A_2) \times A_1)$
- The number of operations (flops) to compute $A_k \times A_{k-1}$ is
  $n_k \times n_{k-1} \times n_{k-2}$
- Cost of forward: $n_0 \sum_{k=1}^{k=t-1} n_k n_{k+1}$
- cost of backward $n_t \sum_{k=0}^{k=t-2} n_k n_{k+1}$

Since, in general, $n_t << n_0$, the backward pass is cheaper

Learning Neural Network with backpropagation

- The cost function is highly non convex, exhibits many local minima, and the number of variables is large
- Tough optimization problem
- Fortunately, optimization can exploit the structure of the network
  - Each training sample is sent through the network and makes prediction: feedforward pass
  - The error is backpropagated to efficiently compute the gradients: backward pass
  - Stochastic or mini batch gradient descent with n epochs (one epoch = one complete pass over all training data)

NN structure



Backpropagation

- Is exactly backward AD mode with special case of NN
- *Old* idea rediscovered in 1986 in Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning representations by back-propagating errors. nature, 323(6088), 533-536.
- Network (parameters w_k) with intermediate variables z_k, y_k computed during forward pass
- the intermediate values are computed for each training sample (input of network) and all values for the batch are kept in memory
- $Z_k = W_k Y_{k-1}$ and $Y_k = f_k(Z_k)$ (pointwise activation function at layer k)
- What we want is the gradient of the loss w.r.t. the internal network parameters θ_k. Usually, the gradients are computed for each training sample and averaged over the batch.

Iterative backpropagation

Iterative backward pass (init with the gradient of the loss w.r.t. the output of the network):

- $$\frac{\partial \mathcal{L}}{\partial z_k(i)} = f_k'(z_k(i)) \frac{\partial \mathcal{L}}{\partial y_k(i)}$$
- $$\frac{\partial \mathcal{L}}{\partial w_k(i,j)} = y_{k-1}(i) \frac{\partial \mathcal{L}}{\partial z_k(j)}$$
- $$\frac{\partial \mathcal{L}}{\partial y_k(i)} = \sum_j w_k(i,j) \frac{\partial \mathcal{L}}{\partial z_k(j)}$$

Standard activation functions and their derivatives

- Logistic $f(x) = \frac{1}{1+e^{-x}}$, leads to $f'(x) = f(x)(1 - f(x))$
- $f(x) = tanh(x)$, $f'(x) = 1 - f^2(x)$
- Relu $f(x) = \mathcal{I}(x > 0)x$, $f'(x) = \mathcal{I}(x > 0)$
- Softplus $f(x) = \log 1 + e^x$, $f'(x) = \frac{1}{1+e^{-x}}$

Now the good news...

- All DL frameworks implement automatic differentiation: you do not need to do the math
- Example in pytorch, as simple as

```
model = MyNeuralNetwork(...)
optimizer = torch.optim.SGD(
  model.parameters(), lr=0.01, momentum=0.9
)

for epoch in range(num_epochs):
    for batch_idx, (features, targets) in enumerate(train_loader):

        forward_pass_outputs = model(features)
        loss = loss_fn(forward_pass_outputs, targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Neural network optimization in practice

Deep learning in practice

- Prepare data: gather, annotate, augmentation, normalize
- Design architecture: layers, components, hyper-parameters
- Define loss function, regularization
- Training: gradient descent, monitor training/validation/test error
- MLops, CI-CD: ensure that the pipeline is robust + evaluate bunch of hyperparameters

Why deep learning works so well?

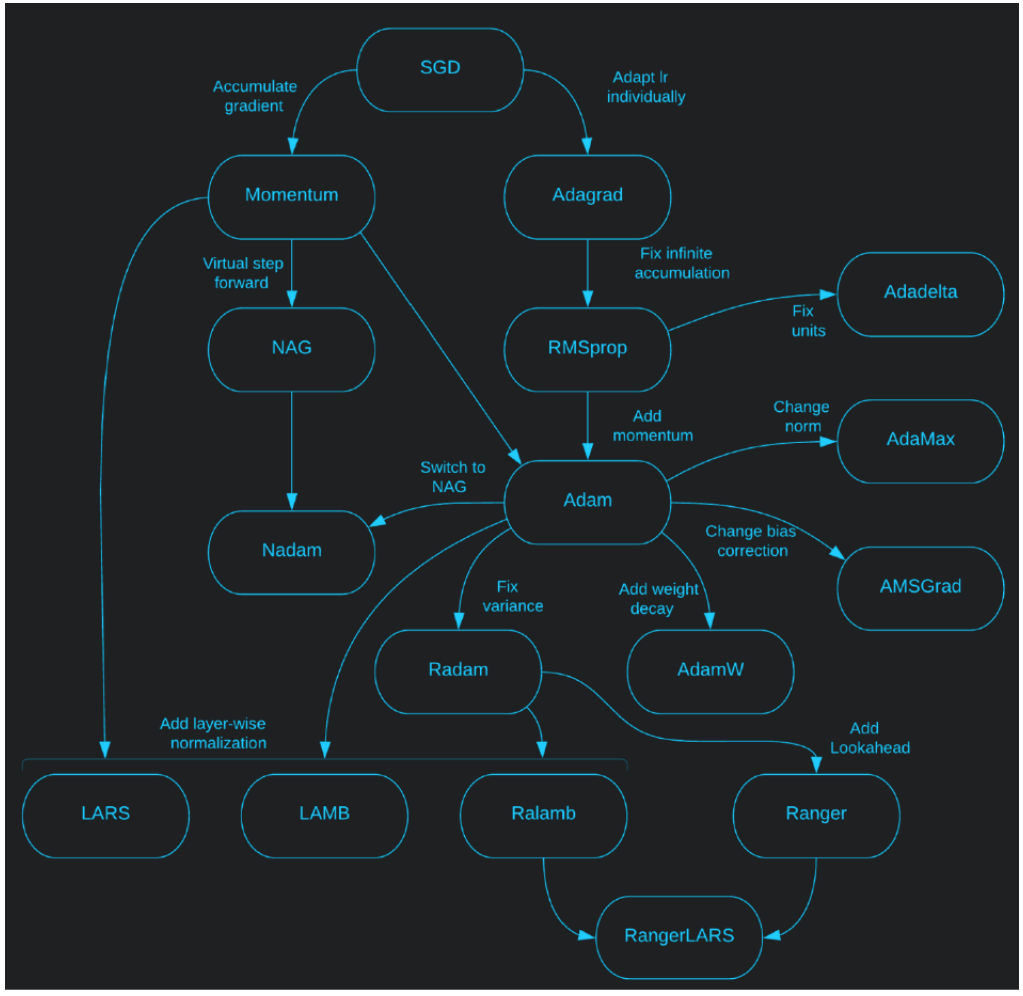
To be honest, there are many unclear factors so far, and there are factors that are problematic:

- Curse of dimensionality: the training dataset is always tiny w.r.t. the possible inputs
- Deep networks describe very complex functions and they generalize surprisingly well with dimension

However, there are ingredients that contribute:

- Efficient optimization algorithms, including gradient clipping
- learning rate scheduler
- initialization
- batchnorm
- Resnets

Optimization algorithms

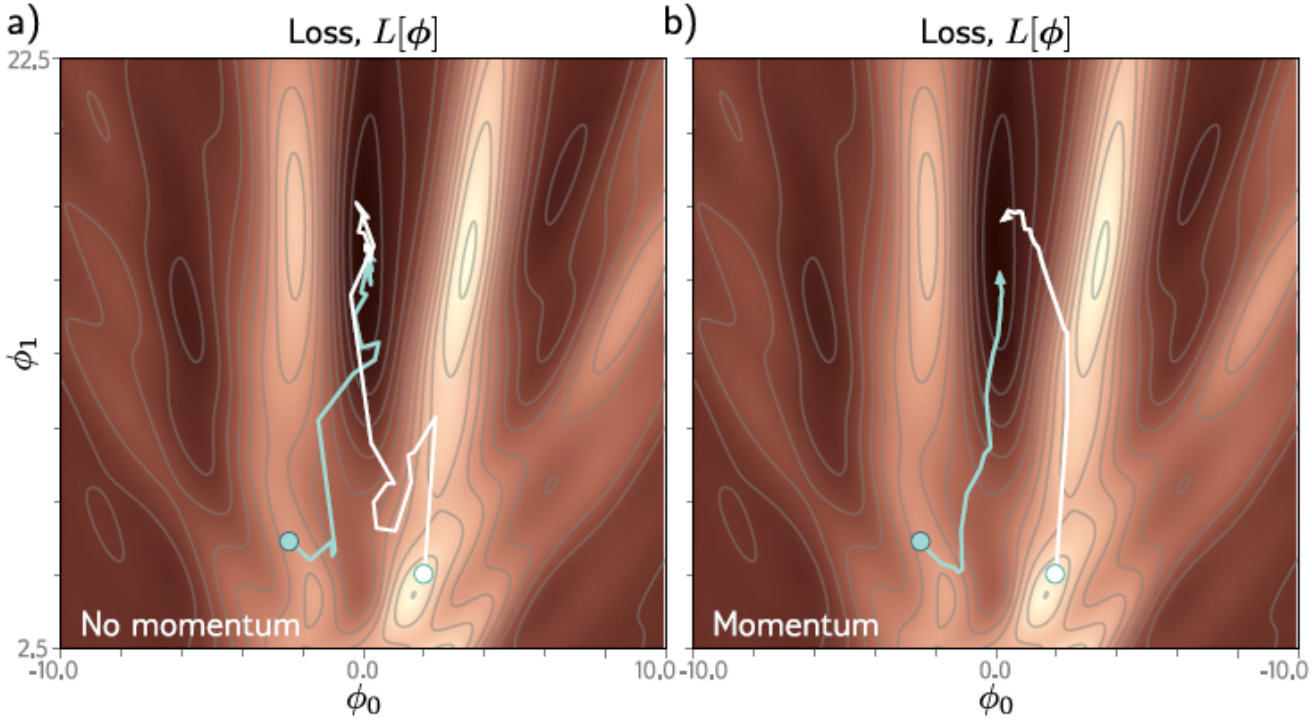


Optimization algorithms

- In practice, only few of them are used and implemented in deep frameworks
- How to pick the best? MLops and accuracy monitoring...
- Popular methods:
  - Momentum methods
  - Adaptive gradient

Momentum methods

- Stochastic gradient descent can be noisy
- Add momentum across batches, with parameter β:
  - $m_{t+1} = \beta m_t + (1-\beta)\sum_{batch} \frac{\partial \mathcal{L}}{\partial \theta}$
  - $\theta_{t+1} = \theta_t - \alpha m_{t+1}$



Adaptive gradients, ADAM

The magnitude of gradients shall be controlled:

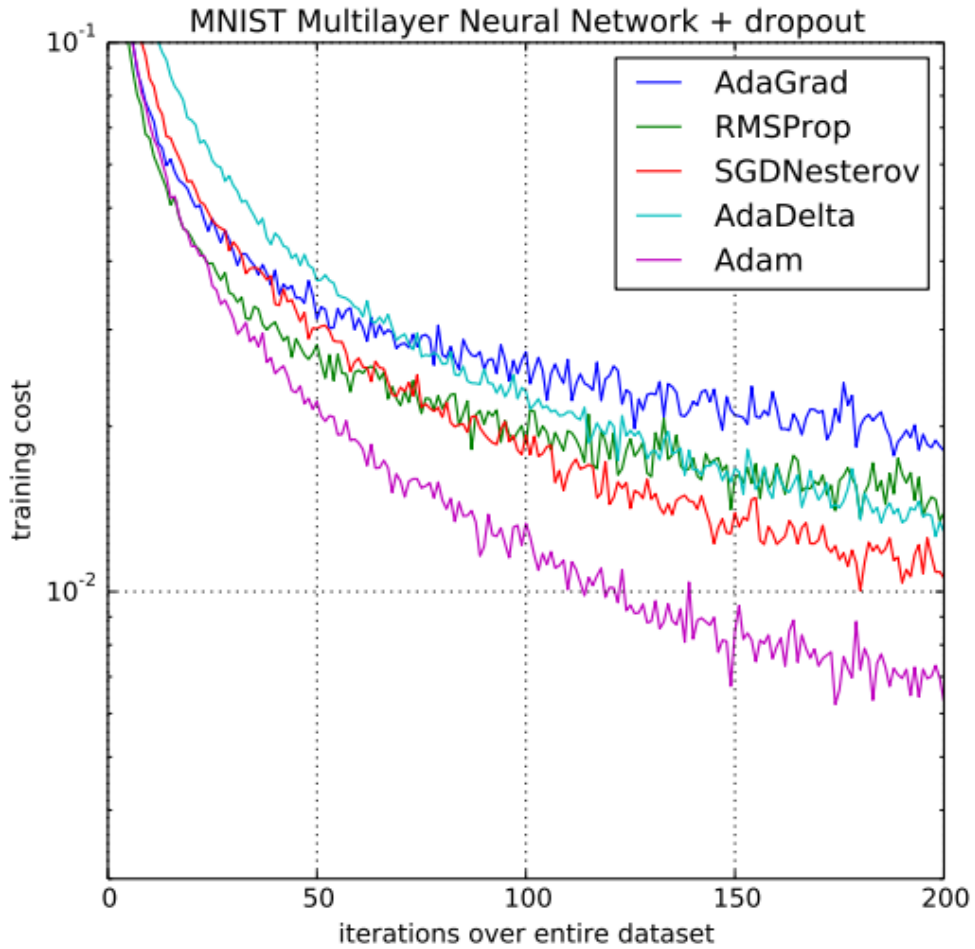
- Small gradients, or vanishing gradients $\Rightarrow$ no learning
- Large gradients, or exploding gradients $\Rightarrow$ instability

Scaled gradients with momentum

- $\theta_{t+1} = \theta_t - G_t^{-\frac{1}{2}} \nabla L$
- exponential moving average $G_t = (1-\beta) \sum_{i=1}^{t} \beta^{t-i} g_i g_i^T$

Visualization of ADAM

KINGMA, Diederik P. et BA, Jimmy. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

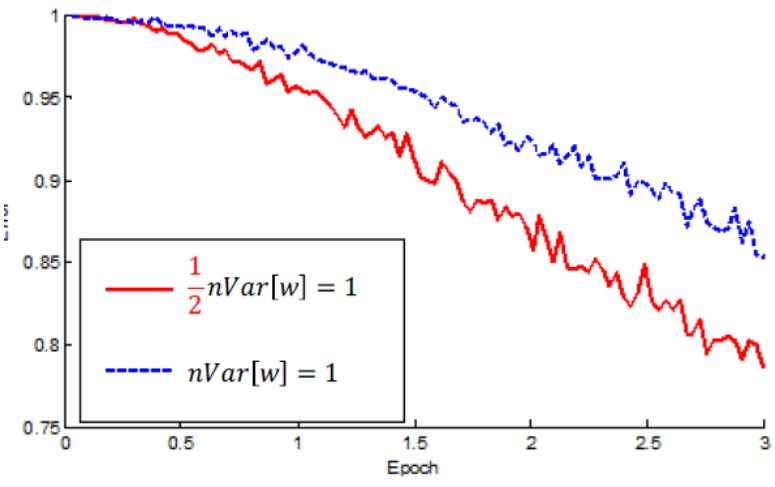


Initialization

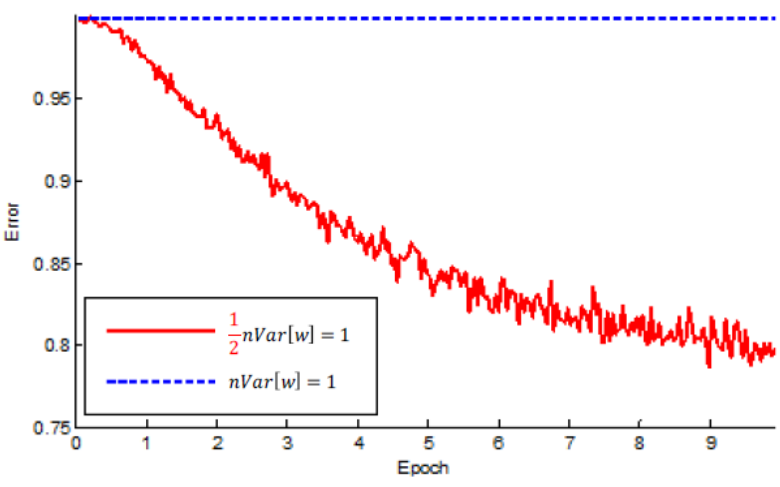
- Since we converge to a local minimum, initialization is crucial
- More in SUTSKEVER, Ilya, MARTENS, James, DAHL, George, et al. On the importance of initialization and momentum in deep learning. In : International conference on machine learning. PMLR, 2013. p. 1139-1147.
- Best strategy: random init to break symmetry. For Relu activations, Gaussian with $\sigma^2 = \frac{2}{n}$

Initialization (2)

22-layer ReLU net:
good init converges faster

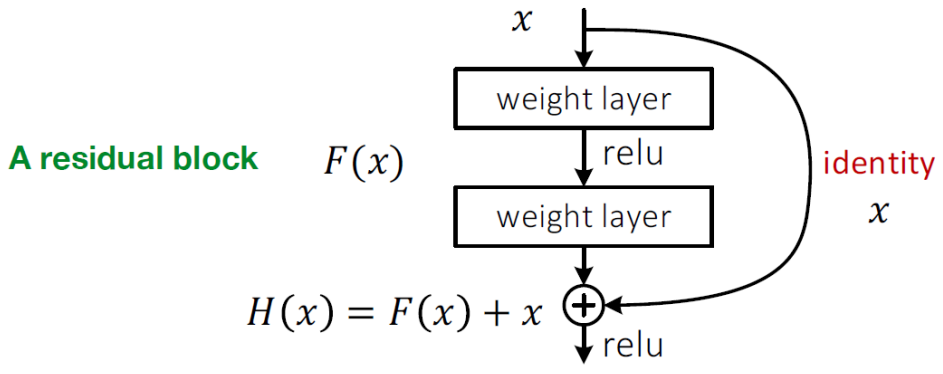


30-layer ReLU net:
good init is able to converge

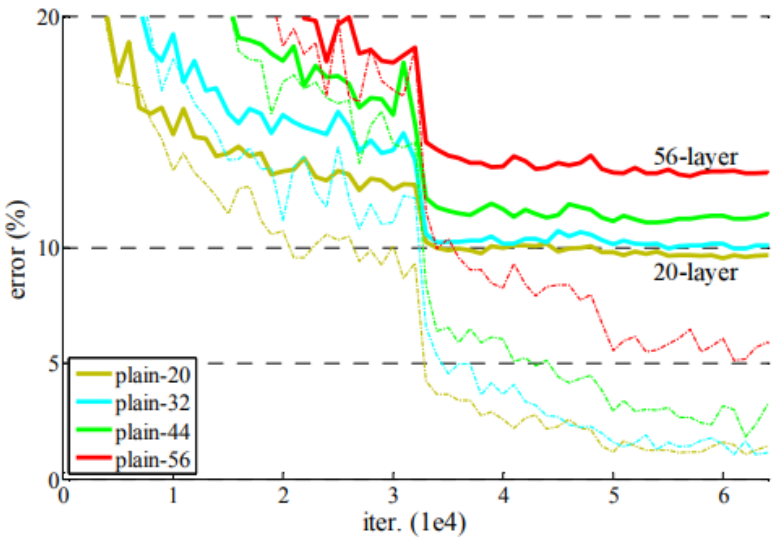


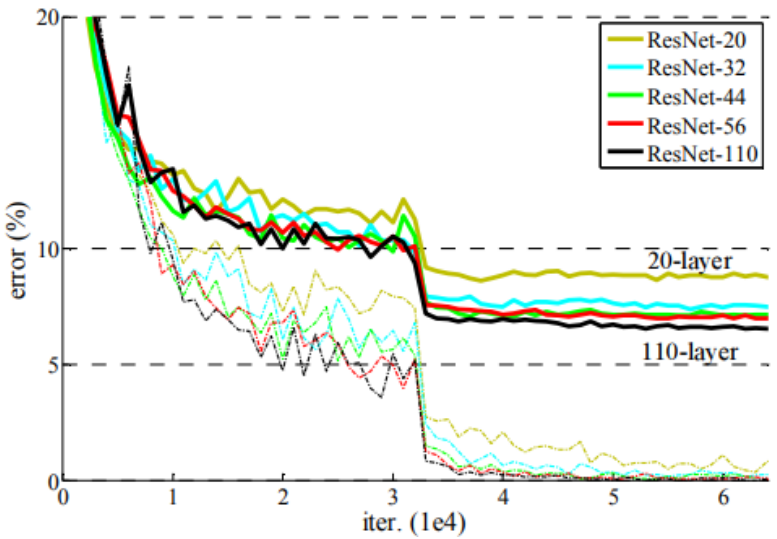
Residual networks

- For ultra deep networks, backprop involves matrix multiplication. If the matrix norms is small, the gradients will exponentially decrease $\Rightarrow$ no learning
- Residual networks ensure that gradients still flow backwards
- More in HE, Kaiming, ZHANG, Xiangyu, REN, Shaoqing, et al. Deep residual learning for image recognition. In : Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. p. 770-778.



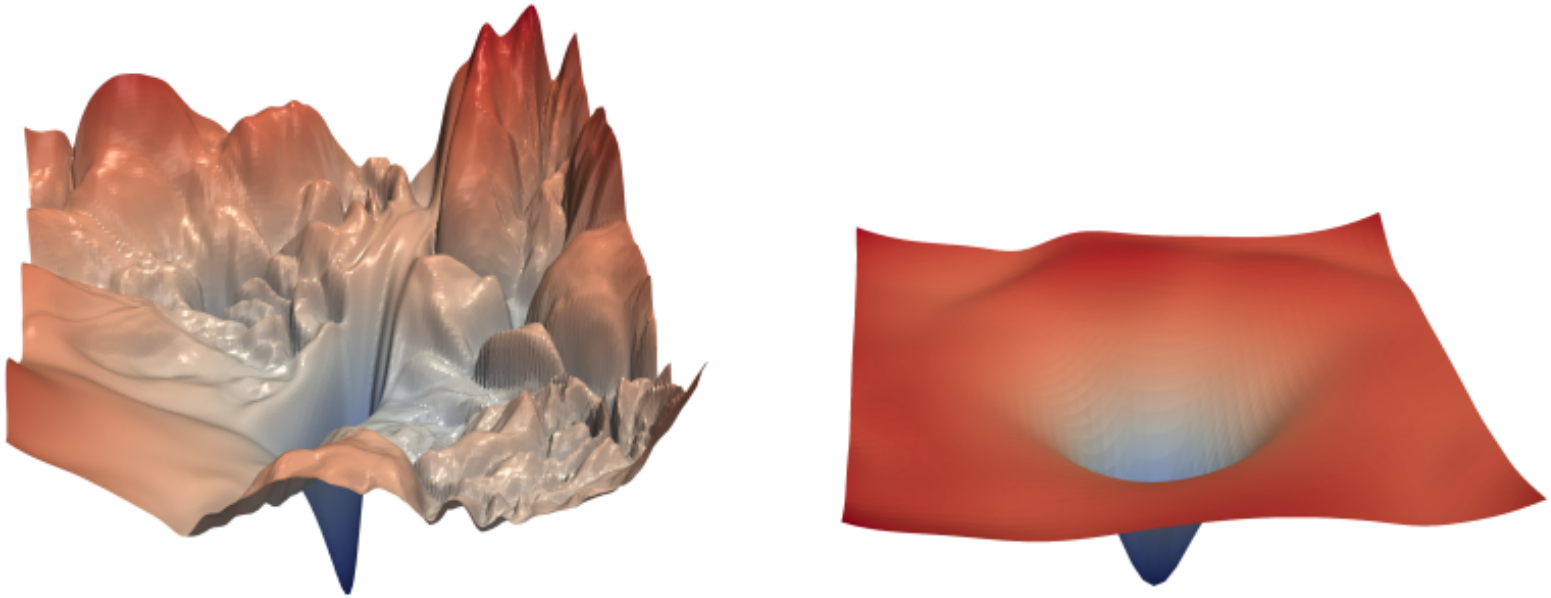
Residual networks (2)





Residual networks (3)

Residual networks convexify the loss landscape (LI, Hao, XU, Zheng, TAYLOR, Gavin, et al. Visualizing the loss landscape of neural nets. Advances in neural information processing systems, 2018, vol. 31.)



(a) without skip connections

(b) with skip connections

Learning rate scheduler

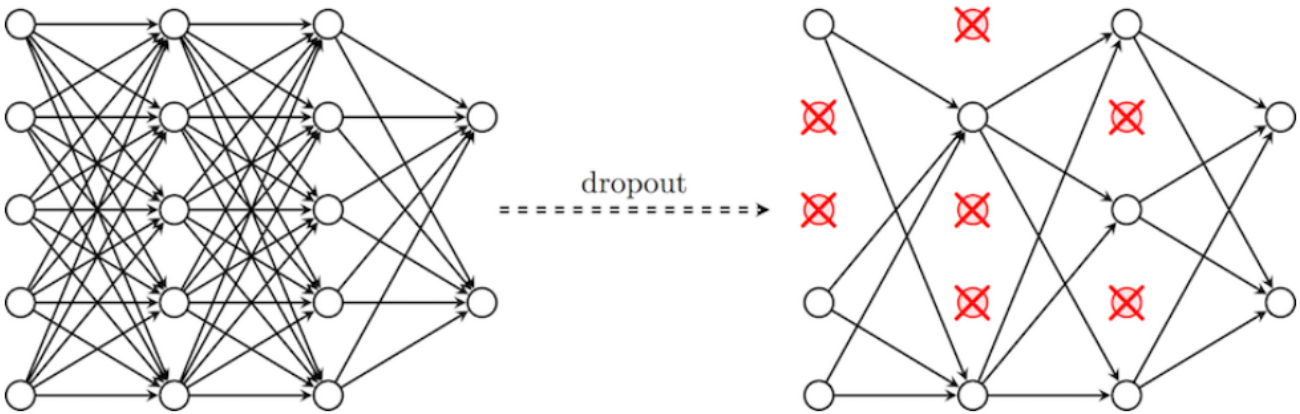
- Tuning the learning rate has a great influence
- Current consensus: learning rate variation over epochs, through scheduler
- Common strategies: time-based decay, step decay, exponential decay
- Implemented in current DL frameworks: *from torch.optim import lr_scheduler*
- Warmup strategies, similar to simulated annealing strategies in the 80s

Learning tricks

- Regularization: penalize with L1 or L2 norm the weights at each layer
- Dropout
- Early stopping
- Data augmentation: increase the number of training data artificially (example: image crop/rotation)

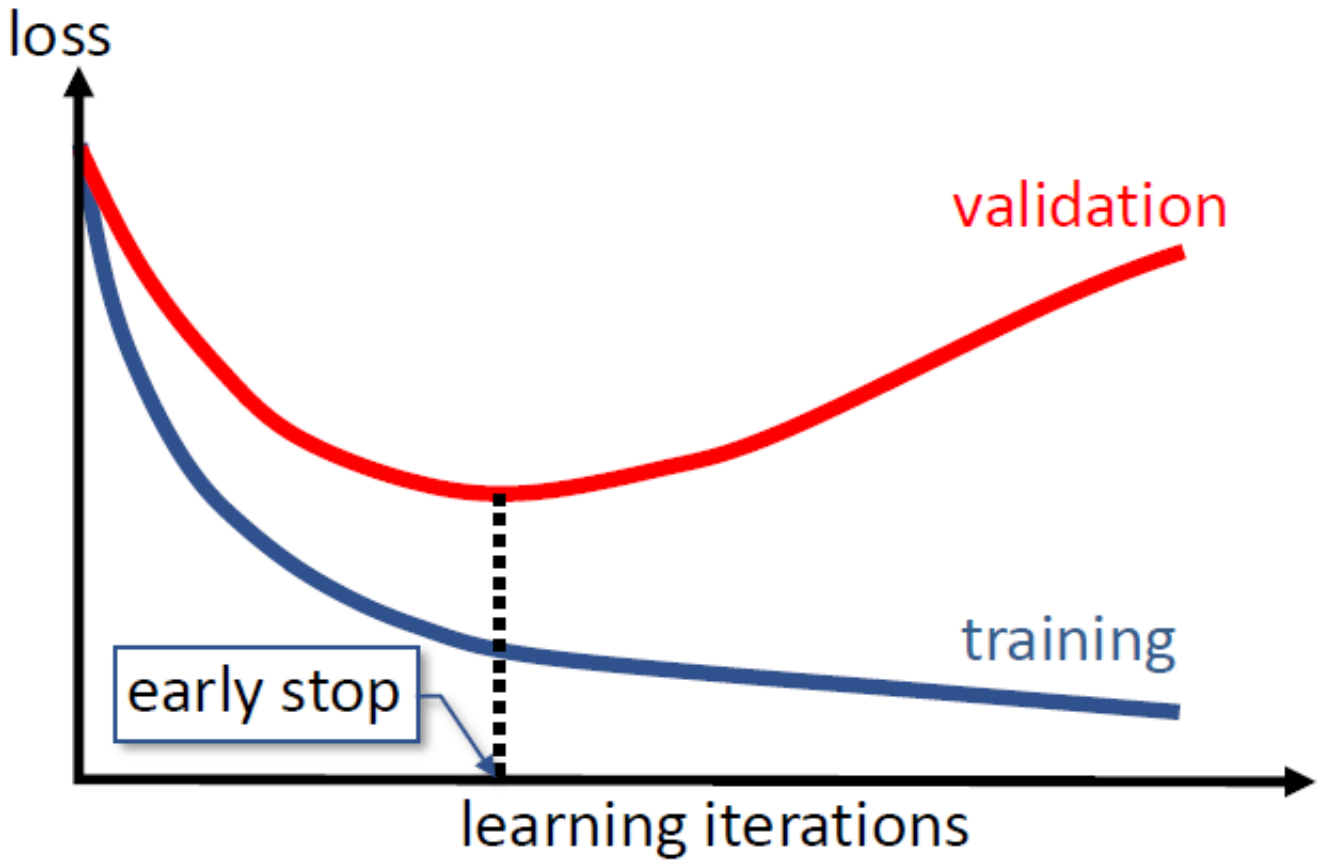
Dropout

- Consists in randomly cutting connections in the network
- Stochasticity helps in regularization



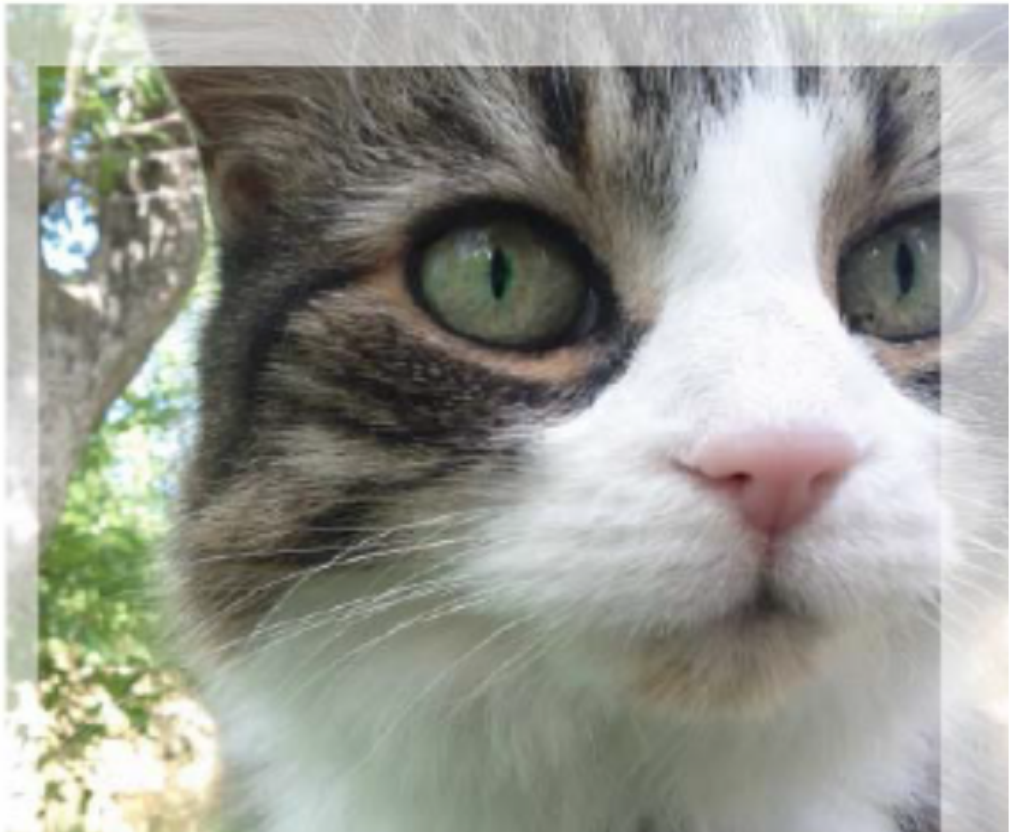
Early stopping

To avoid overfitting, stop the training when validation error increases



Data augmentation

- Increase the volume of training data, crop, flip, rotate, *etc.*
- AlexNet ($\approx$ 60 million parameters) was trained with $\approx$ 1 million data points, with 2048 augmentation per training sample

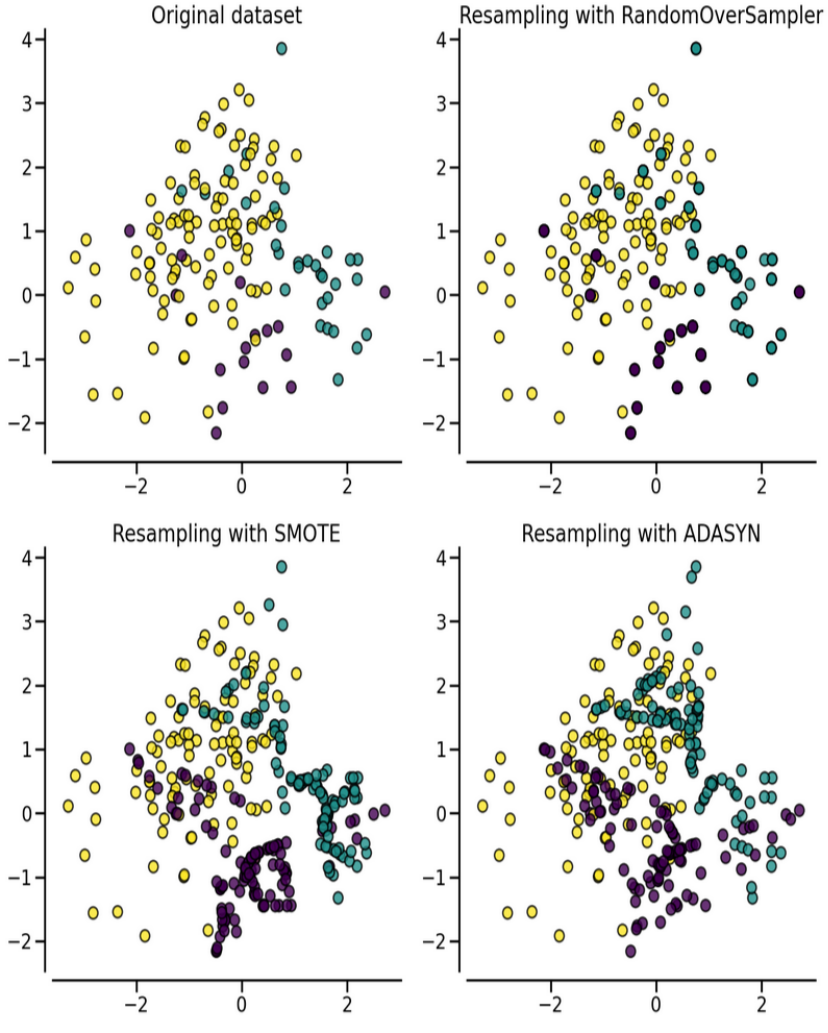


Data bootstrapping

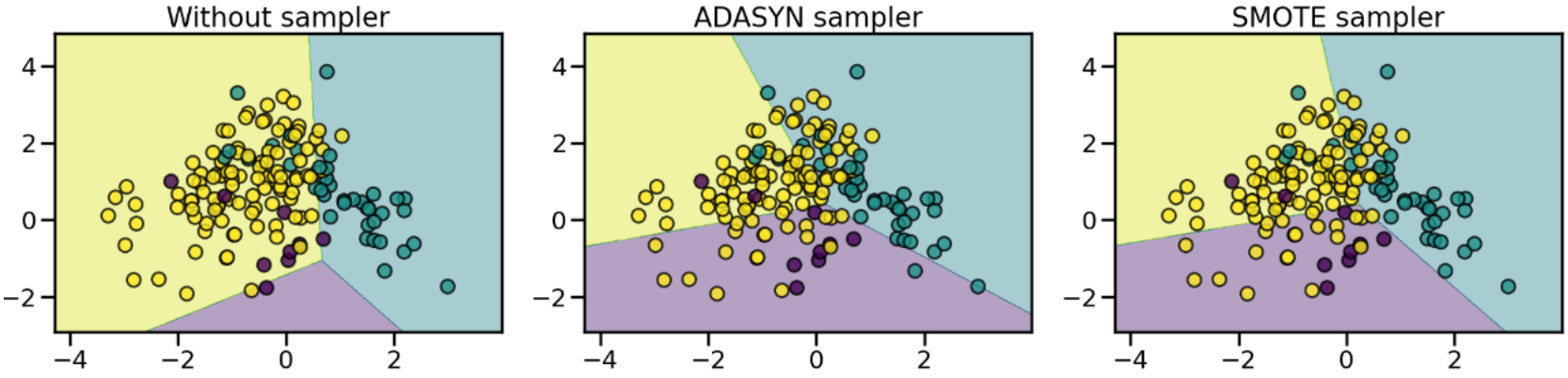
- In many cases, data classes are heavily imbalanced. Examples: medicine, defects, fraud.
- Issue for generalization..
- Solutions:
  - Random (or *"naive"*) oversampling
  - SMOTE (Chawla, Nitesh V., et al. "SMOTE: synthetic minority over-sampling technique." Journal of artificial intelligence research 16 (2002): 321-357.)
  - Adaptive Synthetic (ADASYN) (He, Haibo, et al. "ADASYN: Adaptive synthetic sampling approach for imbalanced learning." 2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence). Ieee, 2008.)

Data synthesis

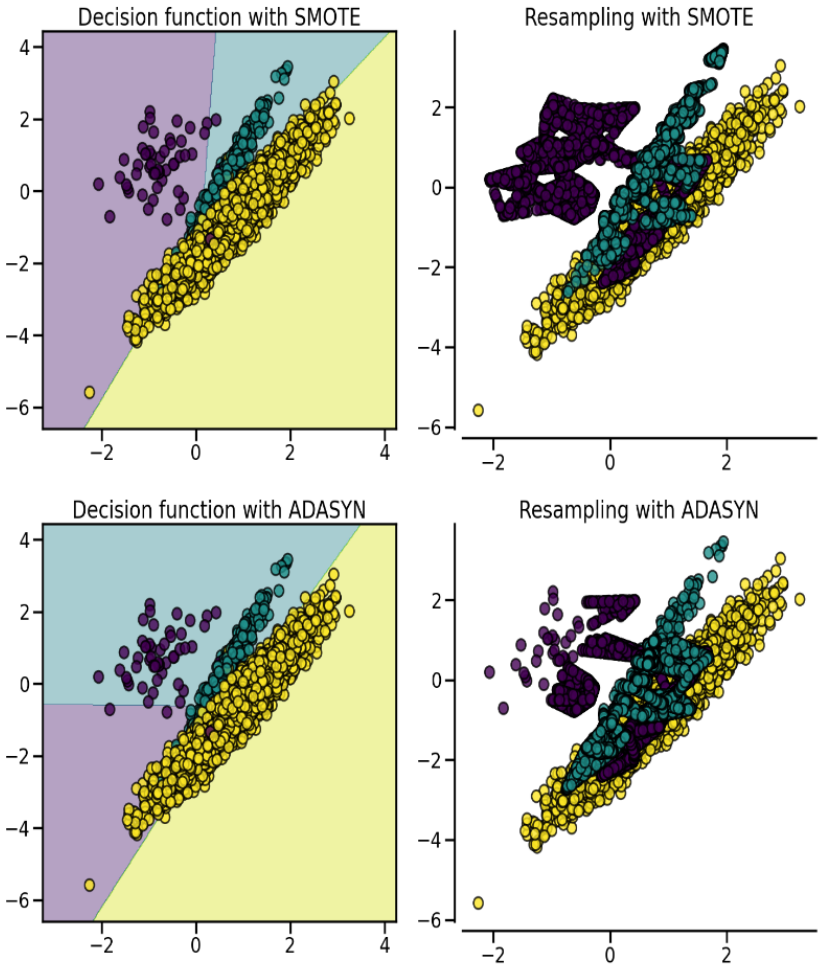
- Smote: upsample minority classes with linear interpolation between sample and nearest neighbors.
- Adasyn: Weighted synthetic data generation, where the weights depend on the difficulty to train for the considered class.



Data synthesis: classification success



Data synthesis: classification issue

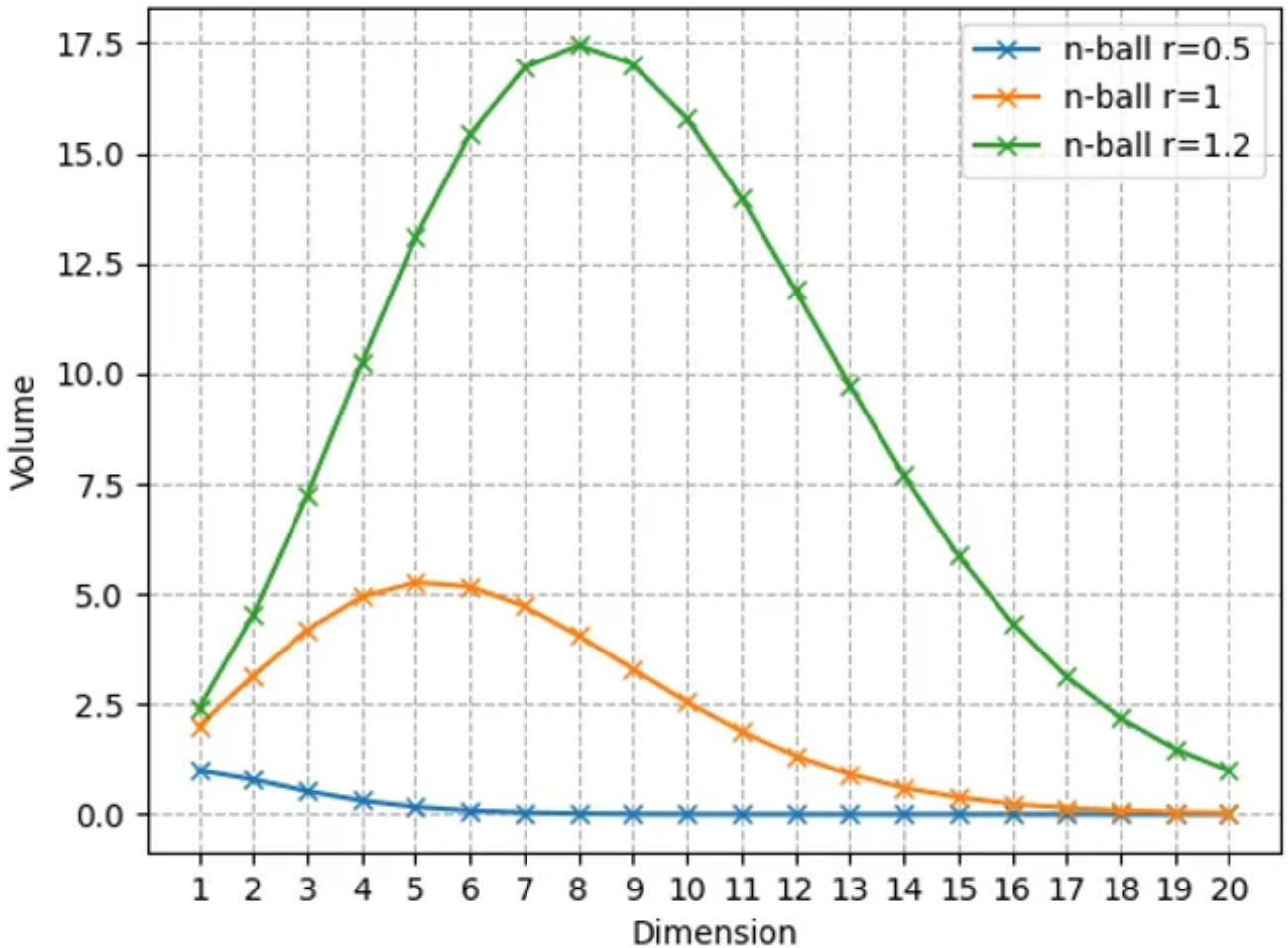


Motivation

- Observations are data in high, or very high dimension
- Data are indirect measurements of an underling source of lower dimension. In other words, the dimensional of data is insanely or ridiculously large
- Dimension reduction: project data to a lower-dimension space
- Benefits of dimension reduction:
  - Data dimension reduction: memory consumption, low-dimensional encoding
  - Data visualization: project data in dimension 2 or 3 for visual assessment
  - Data preprocessing: other tasks (clustering) are simpler in a lower dimension space

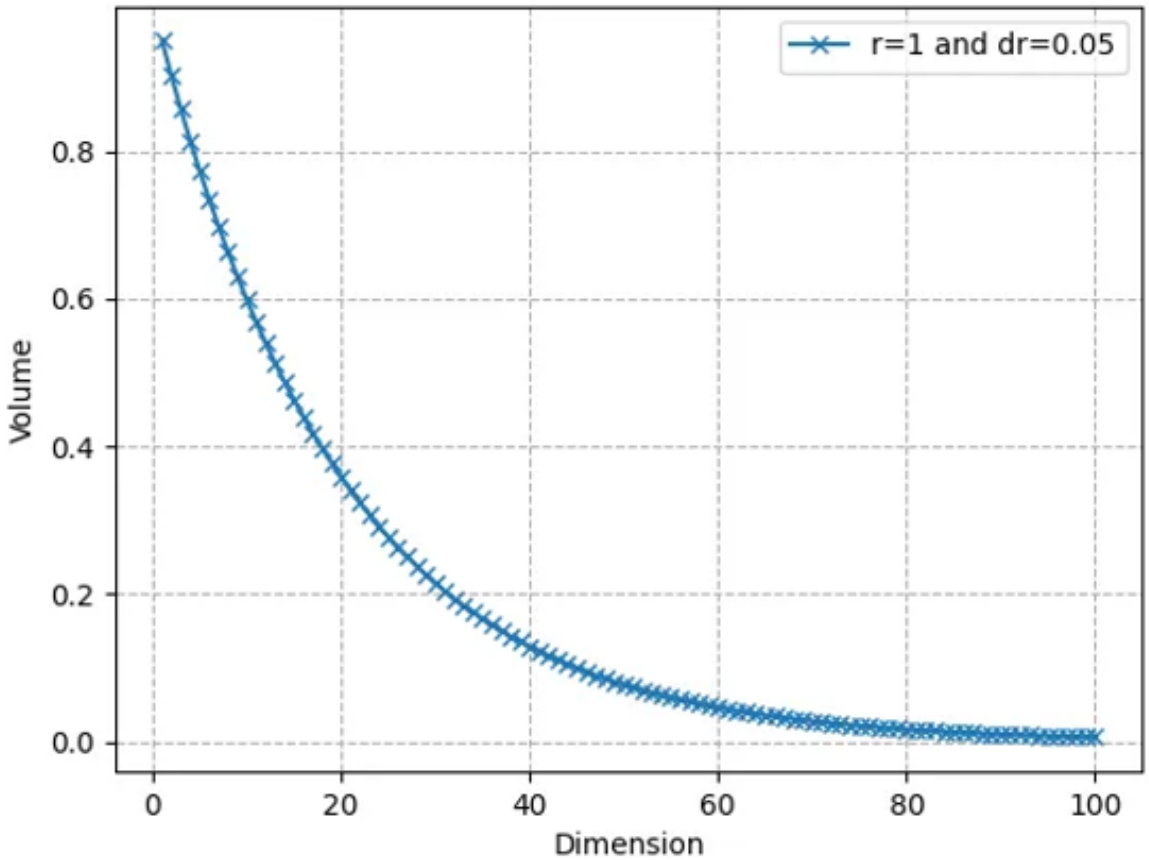
The curse of dimensionality

- Volume of the R-ball in dimension n: $V_n(R) = R^n \frac{\pi^{n/2}}{\Gamma(n/2+1)}$



The curse of dimensionality

- Shell: difference between the R-ball and the (R-dR)-ball, with $dR << R$
- Ratio between interior volume and total volume



Motivation

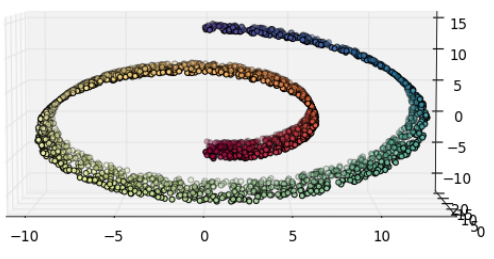
- Observations m points x_i in $\mathbb{R}^n$
- Find projection operator f such that $y_i = f(x_i)$, where $f : \mathbb{R}^n \rightarrow \mathbb{R}^d$, with $d < n$ and possibly $d << n$
- Respect similarities/distances, i.e., $\forall i, \forall j, ||y_i - y_j|| \sim d(x_i, x_j)$
- d is not necessarily an Euclidean distance, but a dissimilarity

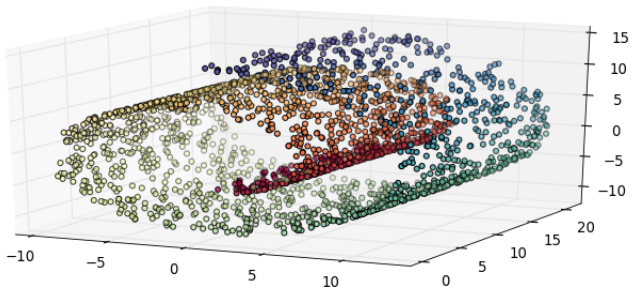
From distance to similarity (and vice versa)

- Distance matrix D $(d_{ij} \sim ||x_i - x_j||)$
- Similarity matrix S $(s_{ij} \sim x_i^t x_j)$
- Distance to similarity by double centering operation $S = (I - \frac{1}{m} 1^t 1) D^2 (I - \frac{1}{m} 1^t 1)$
- Similarity to distance: $d_{ij} = (s_{ii} + s_{jj} - 2s_{ij})^{-\frac{1}{2}}$

Euclidian space and manifolds

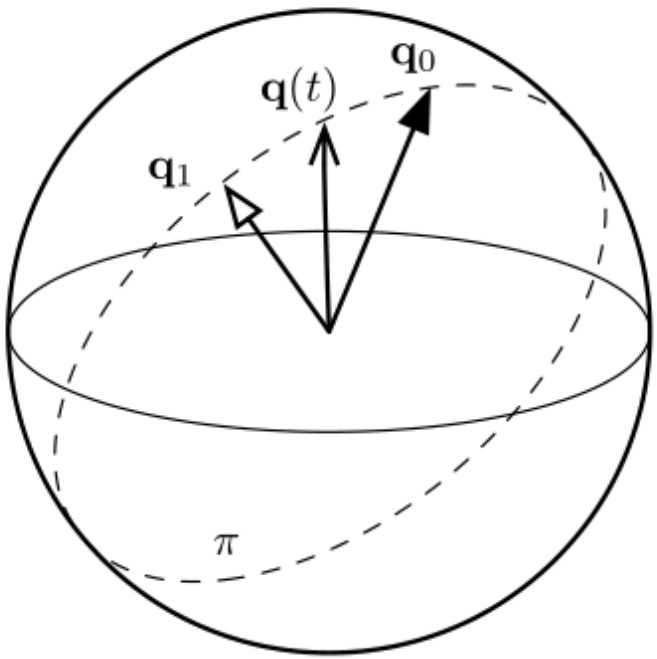
- Metric space: one can compute distances between two points (euclidean distance)
- Manifold: for each point, the neighborhood (or tangent space) is homeomorphic to a subset of the Euclidean space
- The dimension of the manifold is bound to the dimension of the tangent euclidean space
- In other words, the manifold is *locally* Euclidean

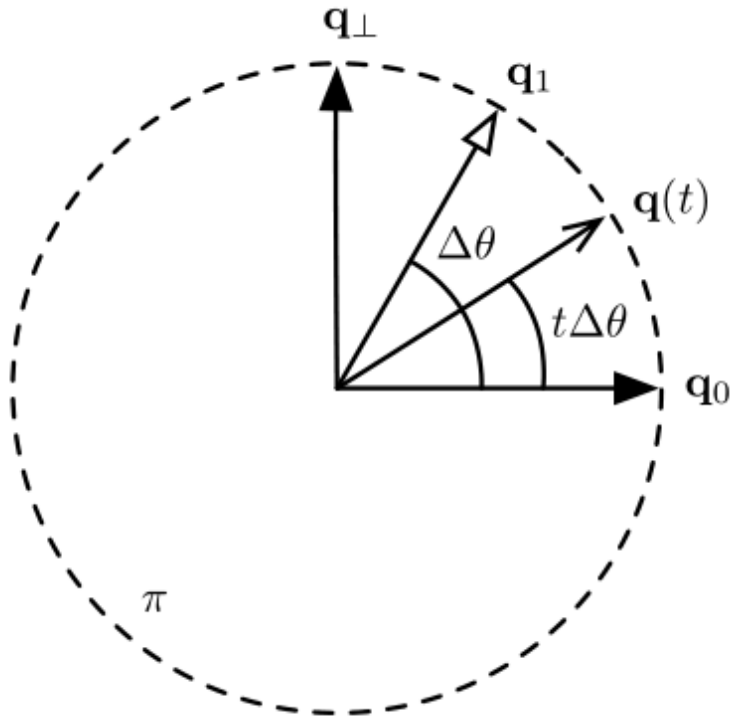




Applications: data visualization

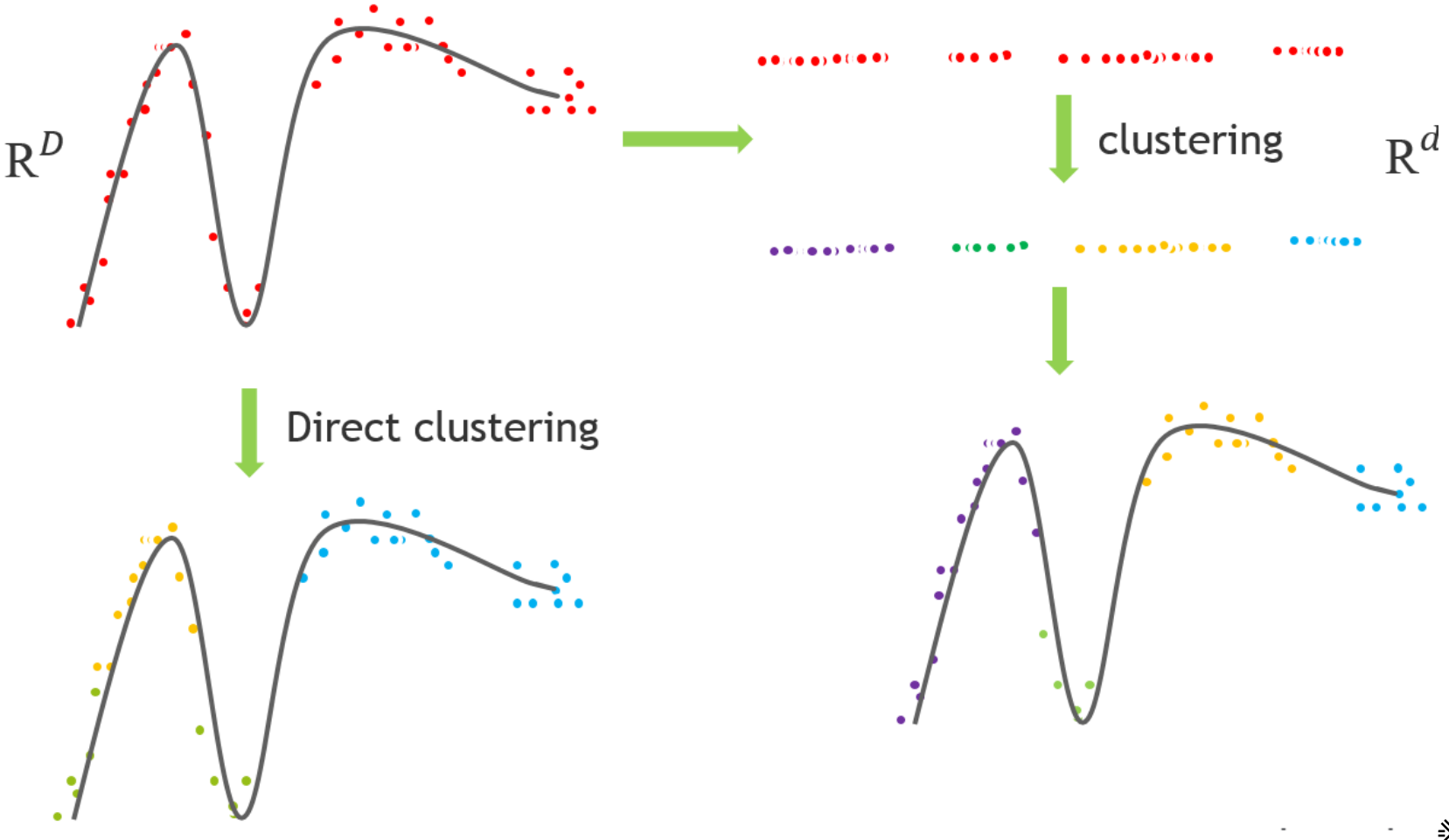
Applications: interpolation





Applications: clustering

- **Data clustering** on the manifold



Linear projection

- PCA is a linear projection operator. Input data $X = [\mathbf{x}_1, \ldots, \mathbf{x}_m] \in \mathbb{R}^{nm}$. Output (or latent) data $Y = [\mathbf{y}_1, \ldots, \mathbf{y}_m] \in \mathbb{R}^{dm}$ with $d < n$
- Projection: $Y = Q^t X$, where $Q^t \in \mathbb{R}^{dn}$
- Reconstruction $\hat{X} = QY$, where $Q \in \mathbb{R}^{nd}$
- Reconstruction can be used for generative models (out of scope)

PCA generalization

- PCA amount to a diagonalization of the covariance matrix
- New basis formed by axes of largest variance
- $\Sigma_X = U \Lambda U^t = \sum_a \lambda_a \mathrm{u_a} \mathrm{u_a}^t$
- Projection in dimension d: $\lambda_1, \ldots, \lambda_d$ are the d largest eigenvalues associated with eigenvectors $\mathrm{u}_1, \ldots, \mathrm{u}_\mathrm{d}$
- $U_d = [\mathrm{u}_1, \ldots, \mathrm{u}_\mathrm{d}] \in \mathbb{R}^{nd}$, $\forall i, \mathrm{y_i} = U_d^t \mathrm{x_i}$
- Projected data is decorrelated (data whitening):

$$\frac{1}{m} \sum_i \mathrm{y_i} \mathrm{y_i}^t = \frac{1}{m} \sum_i U_d^t \mathrm{x_i} \mathrm{x_i}^t U_d = U_d^t \Sigma_X U_d = U_d^t U \Lambda U^t U_d = \Lambda$$

How to choose the dimension d

- The total variance of the data is the trace of the covariance: $tr(\Sigma_X)$
- Generally, d is chosen so as to retain a large proportion of the variance, i.e., 95%
- choose d so that $\frac{\lambda_1+\cdots+\lambda_d}{tr(\Sigma_X)} \approx 0.95$

PCA using Gram matrix

- Under the hypothesis of x_i being Gaussian distributed and independent, PCA is the projection that minimizes the distortion: $\hat{U}^t = arg\min_{U^t \in SO(n)} || \sum_{i=1}^{m} x_i - U^t x_i ||^2$
- Diagonalization of Gram matrix and Covariance matrix can be used: $\Sigma_X = \frac{1}{m} X X^t$ with (λ_i, u_i) and $G = X^t X$ with (μ_i, v_i)

$$XX^t u = m\lambda u \Leftrightarrow X^t X X^t u = m\lambda X^t u$$

Hence, $v = X^t u$ and $\mu = m\lambda$

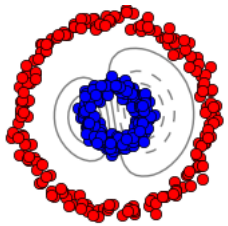
- Interesting if $m < n$ (in high dimension)

Multidimensional scaling (MDS)

- MDS uses the Gram matrix, X is not required (similar to section from data to vectors)
- Goal: compute matrix of embedded data $Y \in \mathbb{R}^{dm}$ such that $\hat{Y} = arg\min_Y ||G - Y^t Y||^2$
- Solution: diagonalization of $G = V\Lambda V^t$, and solution $Y = \Lambda_d^{\frac{1}{2}} V_d^t$, associated to d largest eigenvalues

Motivation

- PCA amount to determining linear projection that maximizes variance of embedded data



- Data is not always linearly separable
- Idea of kernel PCA: find invertible embedding $\Phi : \mathbb{R}^n \to \mathbb{R}^D$, with $D > n$, such that data is linearly separable in $\mathbb{R}^D$.
- Brute force procedure:
  - Compute embedded data, $\forall i, z_i = \Phi(x_i)$
  - Perform PCA on Z
  - Map back with Φ^{-1}
- Issue: complexity and memory since $Cov(Z) \in \mathbb{R}^{DD}$

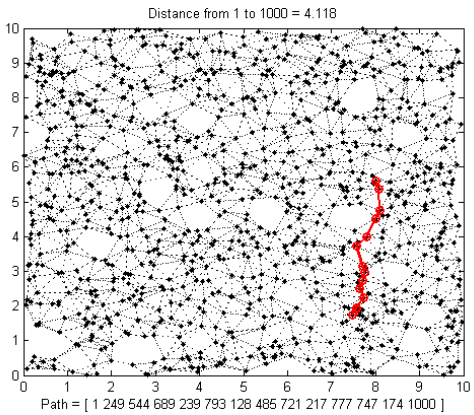
Kernel trick

- However, this can be solved using the kernel trick
- Used in machine learning for a long time (kernel SVM)
- Construct kernel matrix $K = [k_{ij}]$, where $k_{ij} = \Phi(x_i)^t \Phi(x_j)$
- Any algorithm that works with scalar product can be used
- Remember PCA with Gram matrix!
- Popular kernels:
  - RBF: $K(x, y) = e^{\frac{||x-y||^2}{2\sigma^2}}$
  - polynomial: $K(x, y) = (x^t y + \theta)^p$
  - sigmoid: $K(x, y) = tanh(ax^t y + b)$

Isomap motivation

- How to extract non-linear structures?
- After embedding, preserve distances (nearby points remain close)
- PCA/MDS rely on euclidean distance, but geodesic distance matters
- Idea of Isomap:
  - For neighbor points, euclidean distance is fine
  - For distant points, distance is the shortest path on a graph of neighbors

Isomap



The algorithm is composed of the following steps

- Use k-nn technique to find k nearest neighbors (k is a parameter to tune) [Out of scope]
- Compute all distances as shortest paths on graph (Djikstra's algorithm) [Out of scope][a]
- Use MDS on the computed distance matrix [See before]

[a] **dijkstra1971short**.

Isomap pros and cons

Pros:

- preserves the global data structure
- performs global optimization
- non parametric, only neighbor parameter to tune

Cons:

- sensitive to data sampling
- very slow

LLE Motivation

- The data structure is locally linear
- The manifold can be approximated by locally linear patches
- steps of the algorithm:
  1. Compute the k-nn neighbors for each point [out of scope]
  2. Compute the local linear structure
  3. Compute the embedding that preserves the structure

Capture locally linear structure

- Locally linear regression: $\forall i, x_i \approx \sum_{j \in \mathcal{V}_i} w_{ij} x_j$
- Compute weight Matrix

$$\hat{W} = arg \min_W \sum_i ||x_i - \sum_{j \in \mathcal{V}_i} w_{ij} x_j||^2$$

under the constraint $\forall i, \sum_j w_{ij} = 1$

- Invariance to translation:

$$x_i + c - \sum_{j \in \mathcal{V}_i} w_{ij}(x_j + c) = x_i + c - \sum_{j \in \mathcal{V}_i} w_{ij} x_j - c = x_i - \sum_{j \in \mathcal{V}_i} w_{ij} x_j$$

Compute weight matrix

$$\mathcal{C}_i = ||\mathrm{x}_i - \sum_{j \in \mathcal{V}_i} w_{ij} \mathrm{x}_j||^2 = ||\sum_{j \in \mathcal{V}_i} w_{ij} \mathrm{z}_j||^2 = ||w_i^t Z_i||^2$$

with $\mathrm{z}_j = \mathrm{x}_j - \mathrm{x}_i$ locally centered data,
$w_i \in \mathbb{R}^{k1}$ weight vector, where k is the number of neighbors,
$Z_i \in \mathbb{R}^{kn}$ is the local data matrix

Compute weight matrix

$\mathcal{C}_i = w_i^t Z_i Z_i^t w_i$

- $Z_i Z_i^t$ is the Gram matrix of neighbors
- Hence, $\mathcal{C}_i = w_i^t G_i w_i$
- under the constraint $1^t w_i = 1$, lagrange multiplier λ
- $\mathcal{L}(w_i, \lambda) = w_i^t G_i w_i - \lambda(1^t w_i - 1)$
- $\frac{\partial \mathcal{L}}{\partial w_i} = 0 \Leftrightarrow (2G_i w_i - \lambda 1) = 0 \Leftrightarrow w_i = \frac{\lambda}{2} G_i^{-1} 1$
- λ adjusted so that $1^t w_i = 1$
- Gram matrix invertible if $k < n$ (else regularization with $G = G + \alpha I$

Compute embedding

Weights are computed are preserved. Goal: compute $Y \in \mathbb{R}^{dm}$ such that

$$\hat{Y} = arg\min_{Y} \Phi(Y) = \sum_{i} ||y_i - \sum_{j \in \mathcal{V}_i} w_{ij} y_j||^2$$

under the constraints $\forall j, \sum_i y_i = 0$ (data centered) and $\frac{1}{m} Y^t Y = I$ (data decorrelated)

$$\Phi(Y) = \sum_{i=1}^{m} (y_i - \sum_{j \in \mathcal{V}_i} w_{ij} y_j)^2 = ||(I - W)Y||^2 = Y^t M Y$$

with $M = (I - W)^t (I - W) \in \mathbb{R}^{mm}$

Compute embedding

- Adding constraint and Lagrange mutiplier μ:

$$\mathcal{L}(Y, \mu) = Y^t M Y - \mu(\frac{1}{m} Y^t Y - I)$$

  leads to $MY^t = \frac{\mu}{n} Y$
- Similarly, solutions are eigenvectors of M, associated to smallest eigenvalues
- Watch out, trivial solution is vectors of 1, since $M1 = 0$ because $W1 = 1$

LLE pros and cons

Pros:

- can capture highly non linear data
- computation speed
- non parametric, only neighbor parameter to tune

Cons:

- does not preserve the global structure
- sensitive to data sampling

Practical session

Automatic Differentiation on pytorch notebook