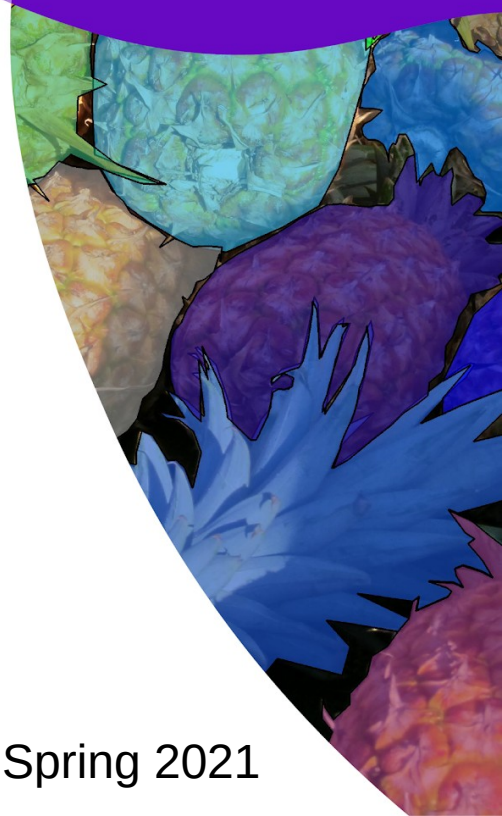
NEW YORK UNIVERSITY

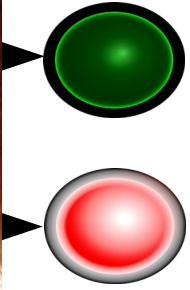
Deep Learning

Alfredo Canziani, Yann LeCun
NYU - Courant Institute & Center for Data Science

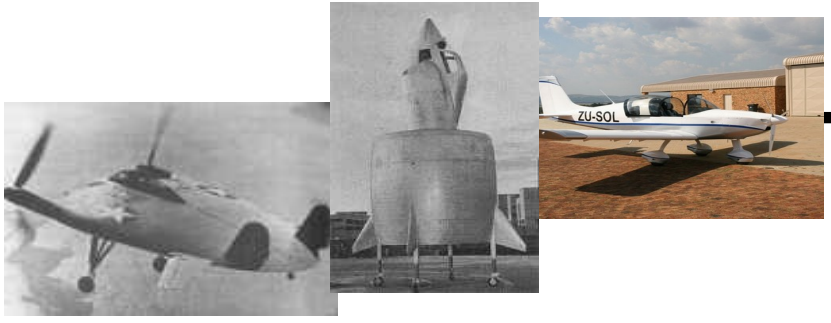


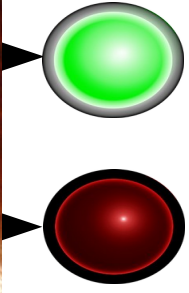
Supervised Learning

- **Training a machine by showing examples instead of programming it**
- **When the output is wrong, tweak the parameters of the machine**
- **Works well for:**
  - Speech → words
  - Image → categories
  - Portrait → name
  - Photo → caption
  - Text → topic
  - ....



CAR

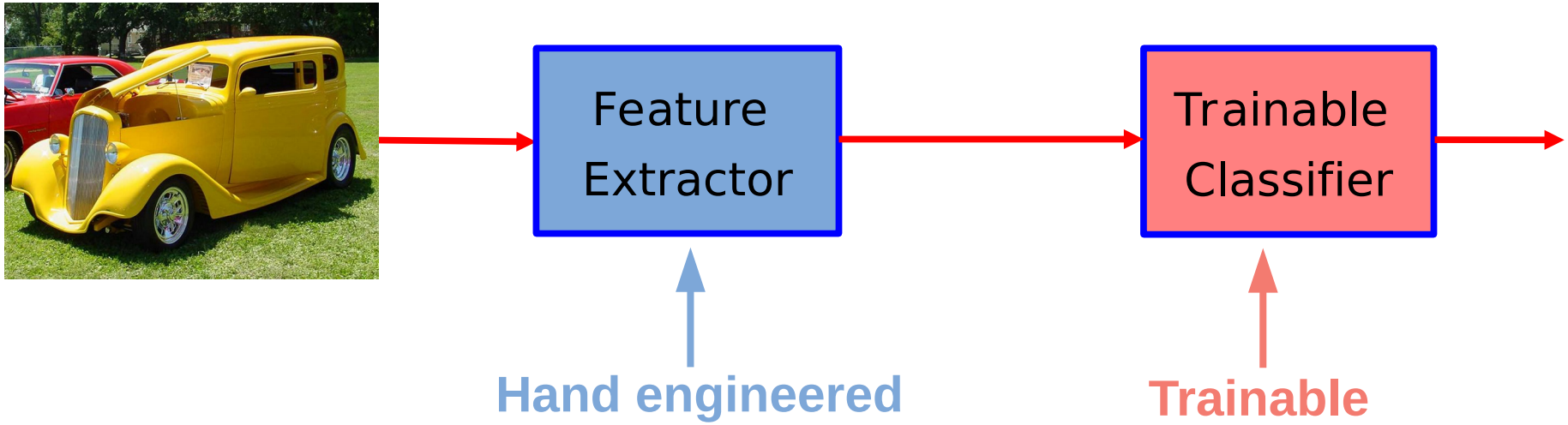




PLANE

The Standard Paradigm of Pattern Recognition

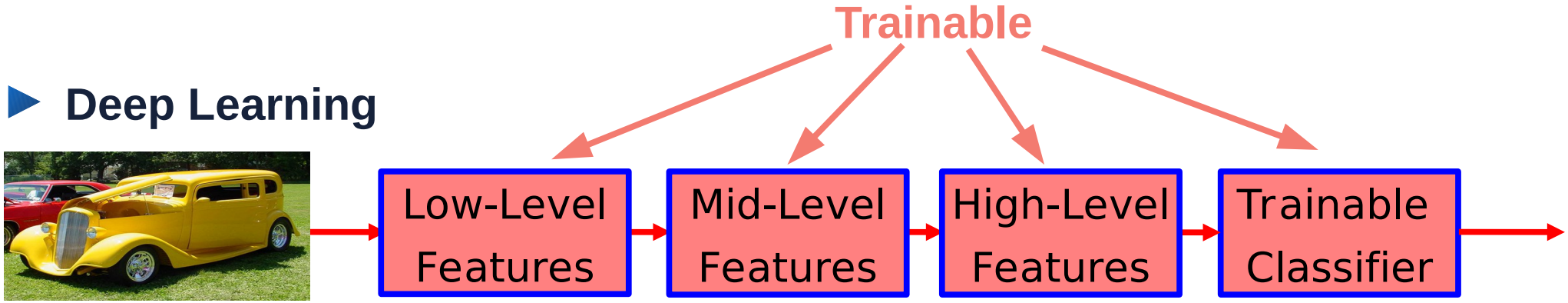
- **...since the 1960s**
- **...and "traditional" Machine Learning**
  - until the "Deep Learning Revolution" (circa 2012)



Multilayer Neural Nets and Deep Learning

- **Traditional Machine Learning**

- **Deep Learning**



Parameterized Model

- **Parameterized model**

$$\bar{y} = G(x, w)$$

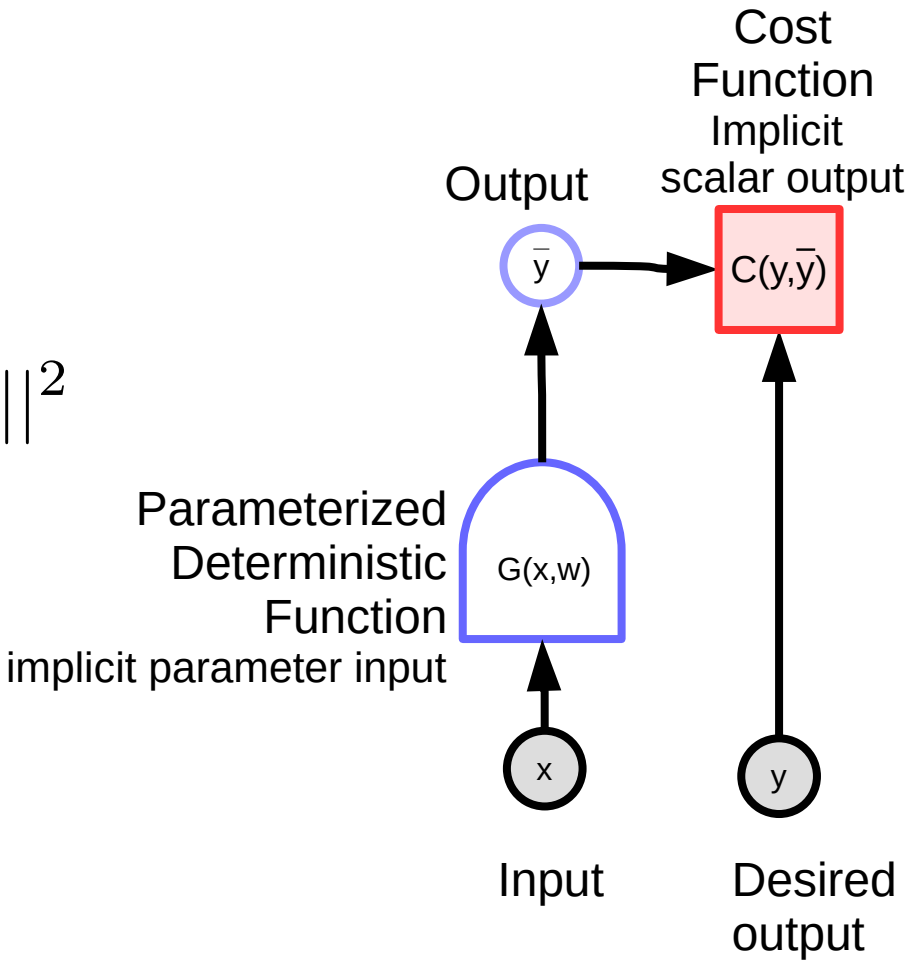
- Example: linear regression

$$\bar{y} = \sum_i w_i x_i \quad C(y, \bar{y}) = ||y - \bar{y}||^2$$

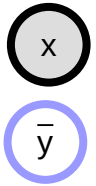
- Example: Nearest neighbor:

$$\bar{y} = \text{argmin}_k ||x - w_{k,.}||^2$$

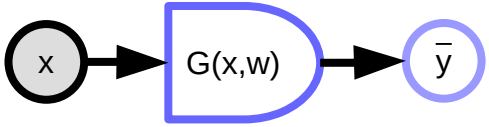
- Computing function G may involve complicated algorithms



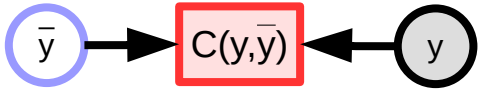
Block diagram notations for computation graphs



- **Variables (tensor, scalar, continuous, discrete...)**
  - Observed: input, desired output…
  - Computed variable: outputs of deterministic functions



- **Deterministic function**
  - Multiple inputs and outputs (tensors, scalars,….)
  - Implicit parameter variable (here: w)



- **Scalar-valued function (implicit output)**
  - Single scalar output (implicit)
  - used mostly for cost functions

Loss function, average loss.

- **Simple per-sample loss function**

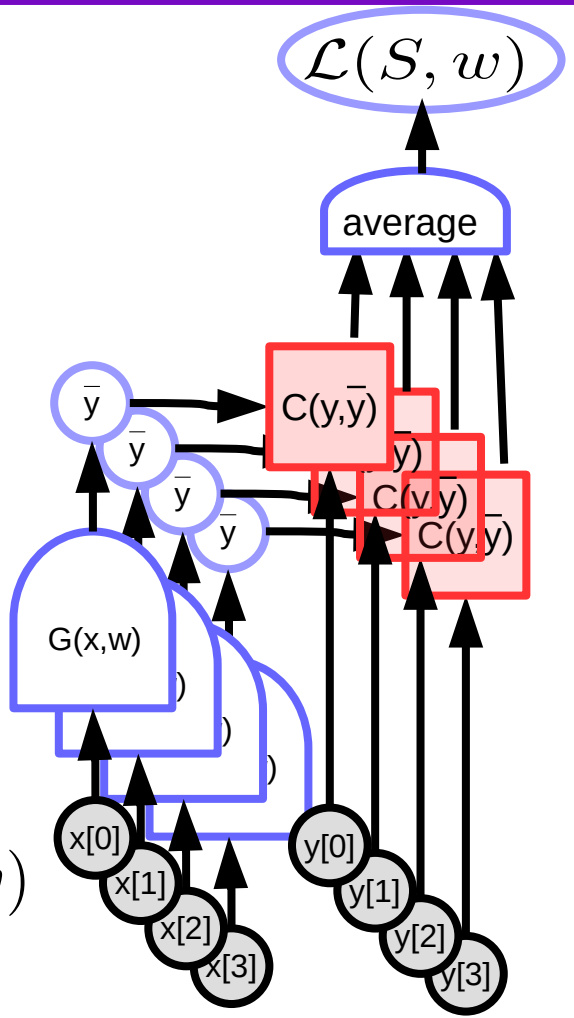
$$L(x, y, w) = C(y, G(x, w))$$

- **A set of samples**

$$S = \{(x[p], y[p]) \ / \ p = 0 \ldots P-1\}$$

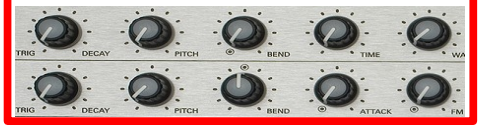
- **Average loss over the set**

$$\mathcal{L}(S, w) = \frac{1}{P} \sum_{(x,y)} L(x, y, w) = \frac{1}{P} \sum_{p=0}^{P-1} L(x[p], y[p], w)$$



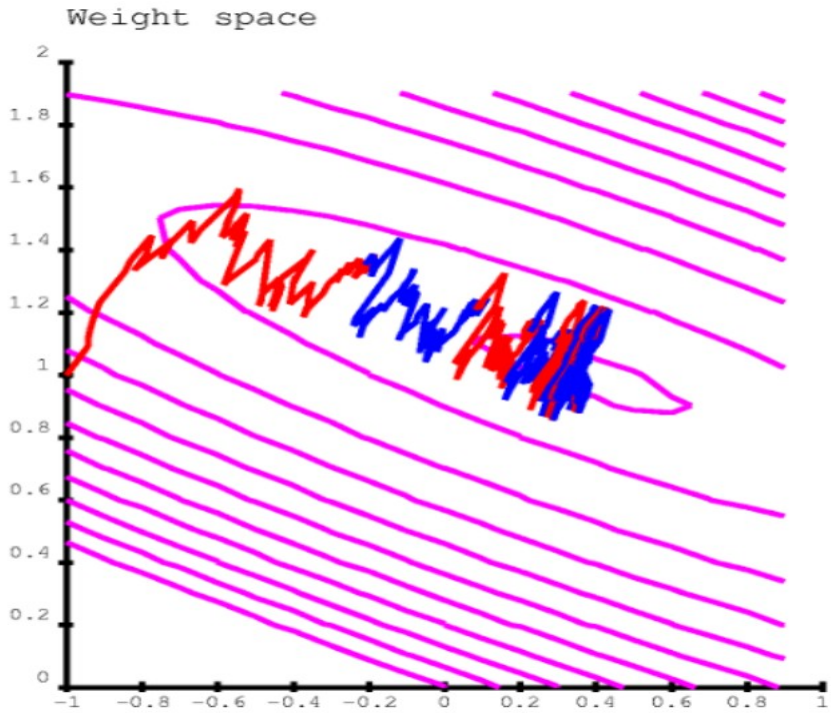
Supervised Machine Learning = Function Optimization

Function with adjustable parameters



Objective Function

traffic light: -1



- **It's like walking in the mountains in a fog and following the direction of steepest descent to reach the village in the valley**
- **But each sample gives us a noisy estimate of the direction. So our path is a bit random.**

$$W_i \leftarrow W_i - \eta \frac{\partial L(W, X)}{\partial W_i}$$

Gradient Descent

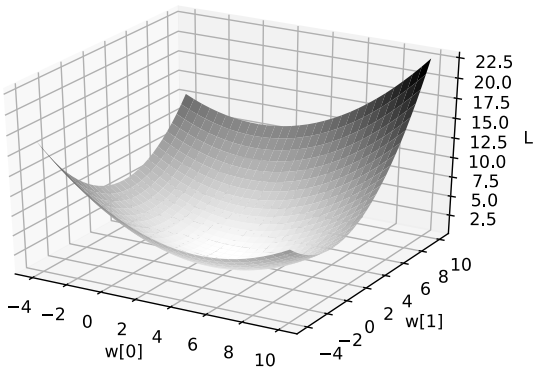
- **Full (batch) gradient**

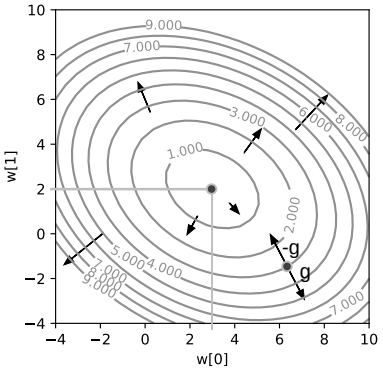
$$w \leftarrow w - \eta \frac{\partial \mathcal{L}(S, w)}{\partial w}$$

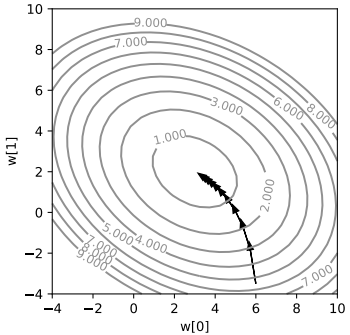
- **Stochastic Gradient (SGD)**
  - Pick a p in 0...P-1, then update w:

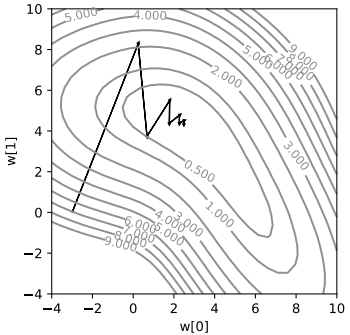
$$w \leftarrow w - \eta \frac{\partial L(x[p], y[p], w)}{\partial w}$$

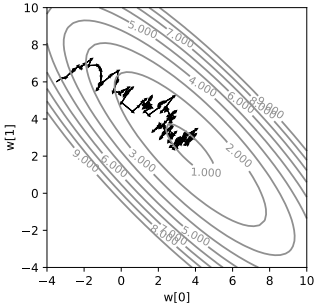
- **SGD exploits the redundancy in the samples**
  - It goes faster than full gradient in most cases
  - In practice, we use mini-batches for parallelization.





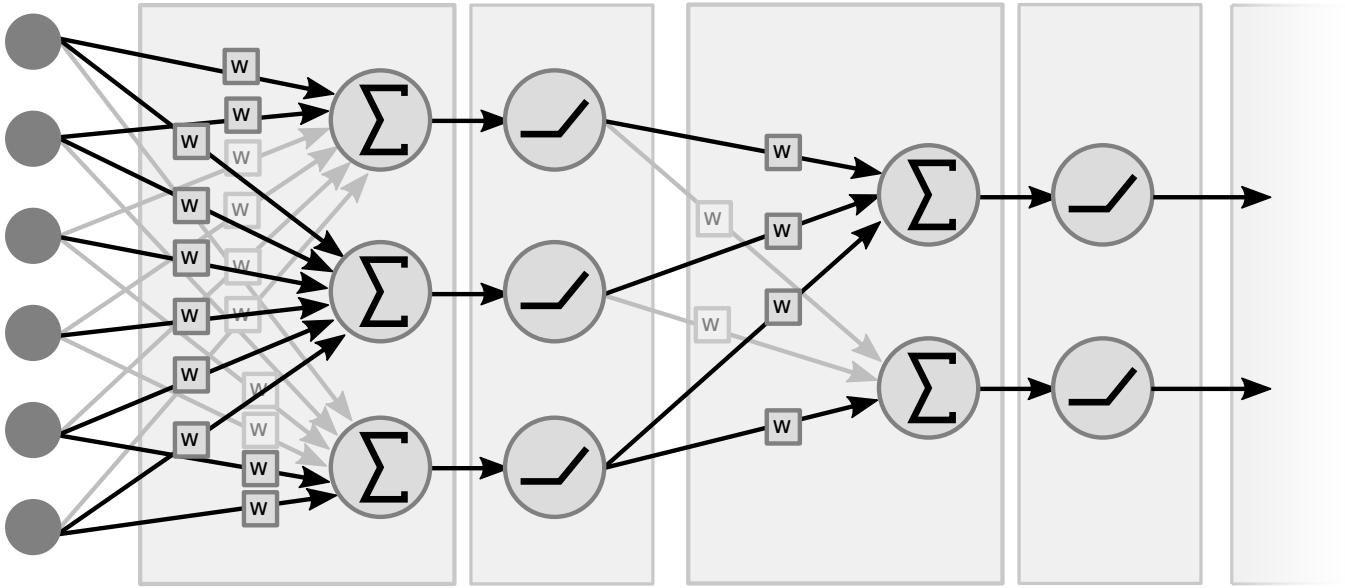






Traditional Neural Net

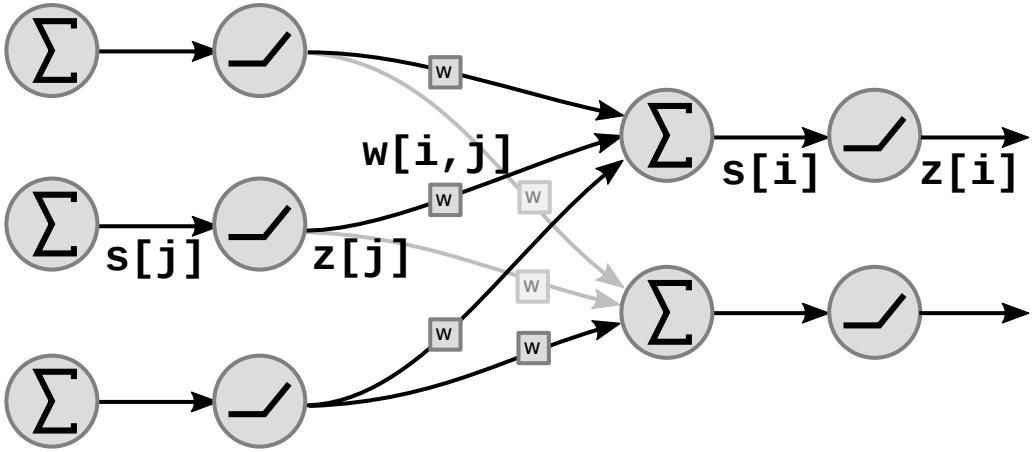
- **Stacked linear and non-linear functional blocks**
  - Weighted sums, matrix-vector product
  - Point-wise non-linearities (e.g. ReLu, tanh, ….)



Traditional Neural Net

- **Stacked linear and non-linear functional blocks**

$$s[i] = \sum_{j \in \mathrm{UP}(i)} w[i,j] \cdot z[j] \qquad z[i] = f(s[i])$$



Backprop through a non-linear function

- **Chain rule:**

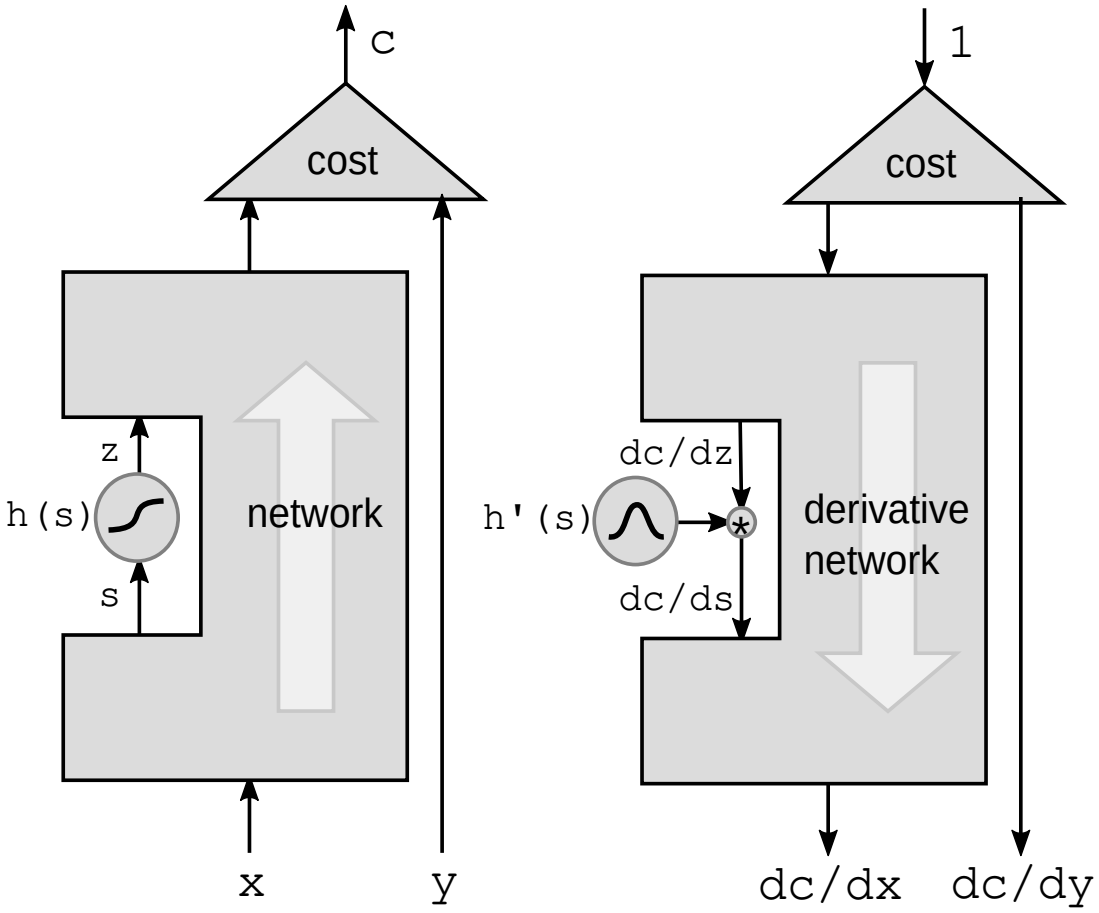
  g(h(s))' = g'(h(s)).h'(s)

  dc/ds = dc/dz*dz/ds

  dc/ds = dc/dz*h'(s)

- **Perturbations:**
  - Perturbing s by ds will perturb z by: dz=ds*h'(s)
  - This will perturb c by

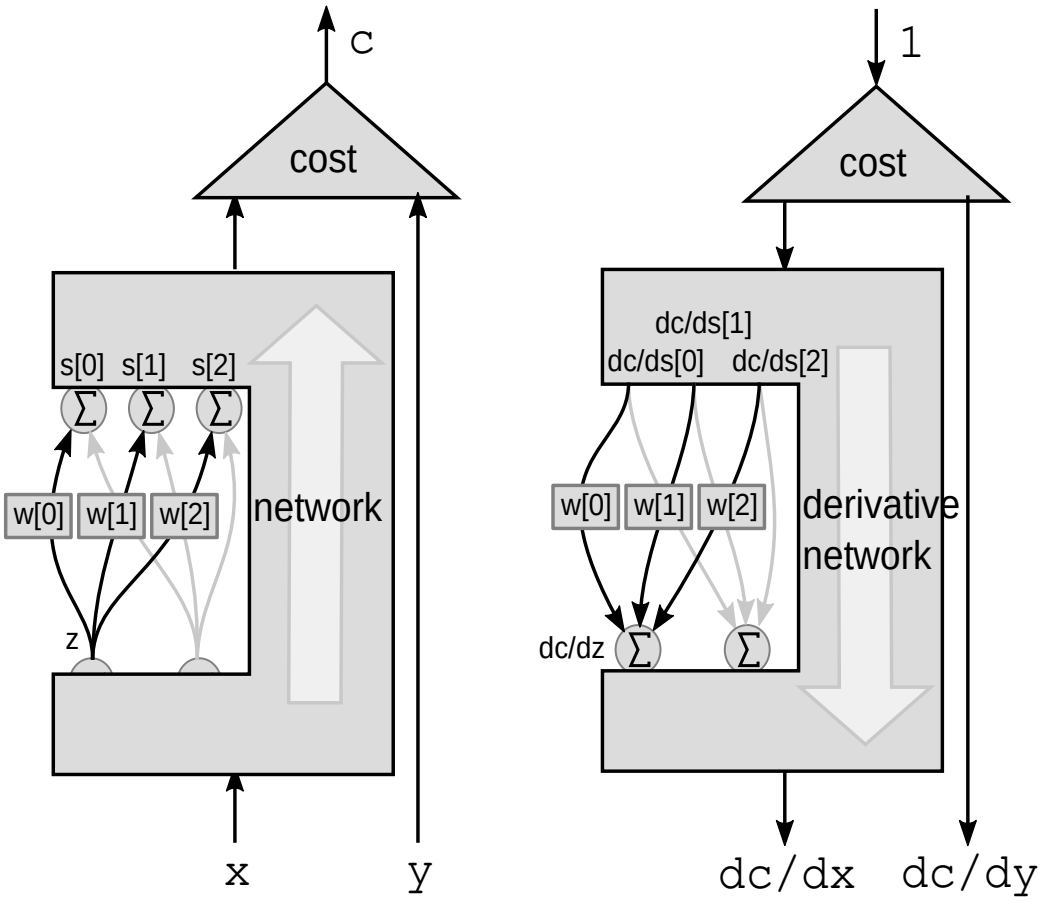
    dc = dz*dc/dz = ds*h'(s)*dc/dz
  - Hence: dc/ds = dc/dz*h'(s)



Backprop through a weighted sum

- **Perturbations:**
  - Perturbing z by dz will perturb s[0],s[1],s[2] by ds[0]=w[0]*dz, ds[1]=w[1]*dz, ds[2]=w[2]*dz
  - This will perturb c by

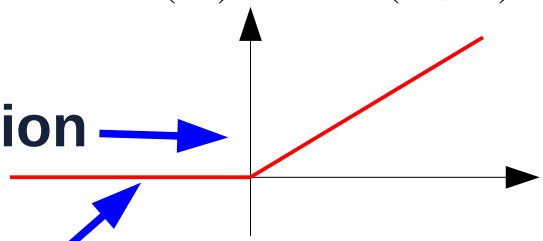
    dc = ds[0]*dc/ds[0]+
    ds[1]*dc/ds[1]+
    ds[2]*dc/ds[2]
  - Hence: dc/dz = dc/ds[0]*w[0]+
    dc/ds[1]*w[1]+
    dc/ds[2]*w[2]+



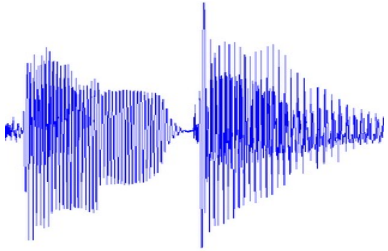
(Deep) Multi-Layer Neural Nets

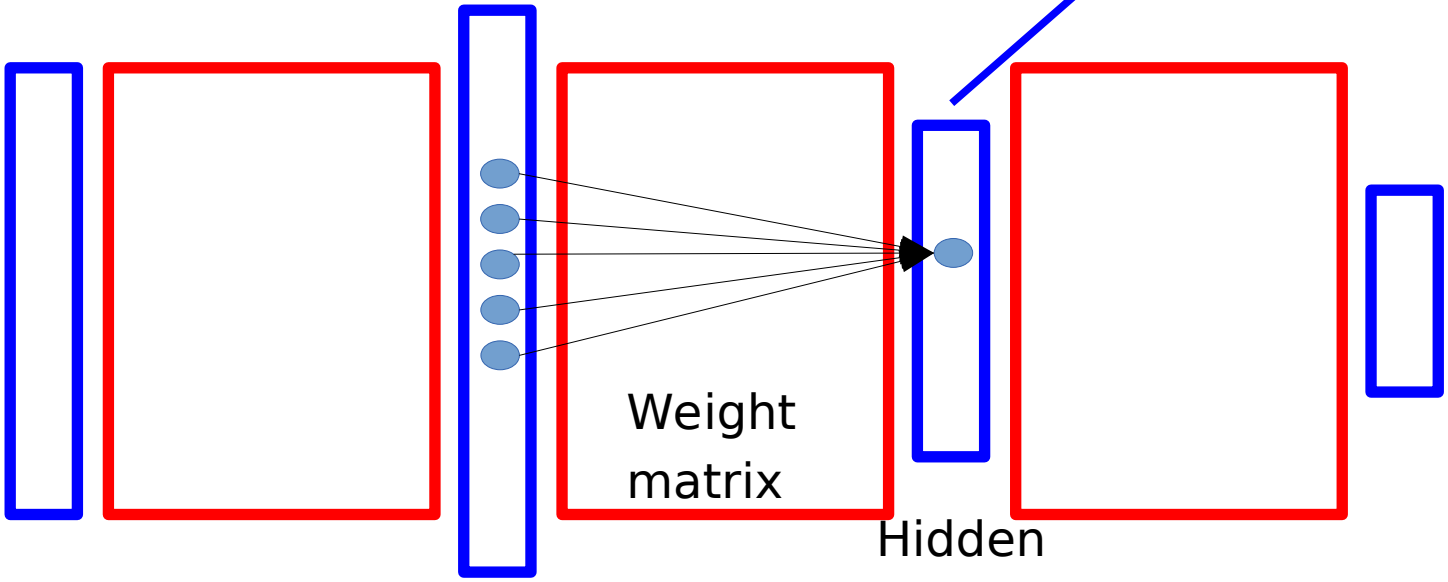
- Multiple Layers of simple units
- Each units computes a weighted sum of its inputs
- Weighted sum is passed through a non-linear function
- The learning algorithm changes the weights

$$ReLU(x) = max(x, 0)$$



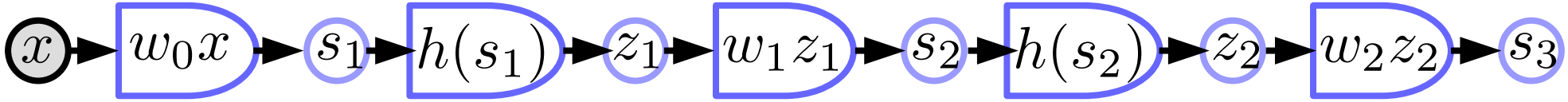
Ceci est une voiture





Block Diagram of a Traditional Neural Net

- **linear blocks** $$s_{k+1} = w_k z_k$$
- **Non-linear blocks** $$z_k = h(s_k)$$



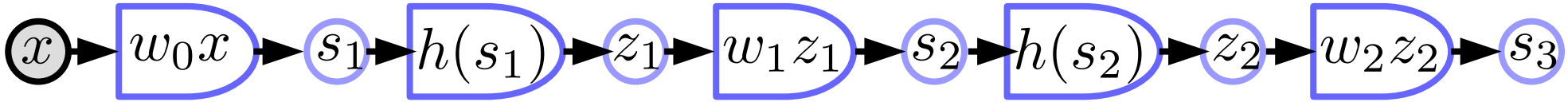
PyTorch definition

- **Object-oriented version**
  - Uses predefined nn.Linear class, (which includes a bias vector)
  - Uses torch.relu function
  - State variables are temporary

```python
import torch
from torch import nn
image = torch.randn(3, 10, 20)
d0 = image.nelement()

class mynet(nn.Module):
    def __init__(self, d0,d1,d2,d3):
        super().__init__()
        self.m0 = nn.Linear(d0, d1)
        self.m1 = nn.Linear(d1, d2)
        self.m2 = nn.Linear(d2, d3)
    def forward(self, x):
        z0 = x.view(-1)  ## flatten input tensor
        s1 = self.m0(x)
        z1 = torch.relu(s1)
        s2 = self.m1(z1)
        z2 = torch.relu(s2)
        s3 = self.m2(z2)
        return s3

model = mynet(d0,60,40,10)
out = model(image)
```



Backprop through a functional module

- **Using chain rule for vector functions**

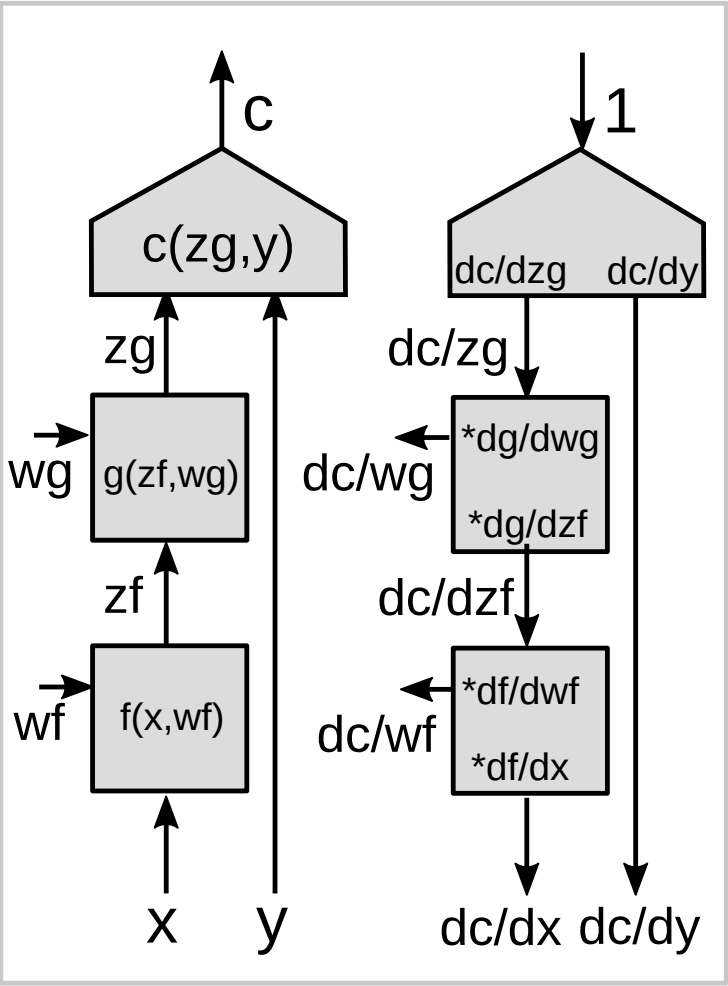
$$z_g : [d_g \times 1] \quad z_f : [d_f \times 1]$$

$$\frac{\partial c}{\partial z_f} = \frac{\partial c}{\boxed{\partial z_g}} \boxed{\partial z_g} \frac{}{\partial z_f}$$

$$[1 \times d_f] = [1 \times d_g] * [d_g \times d_f]$$

- **Jacobian matrix**
  - Partial derivative of i-th output w.r.t. j-th input

$$\left(\frac{\partial z_g}{\partial z_f}\right)_{ij} = \frac{(\partial z_g)_i}{(\partial z_f)_j}$$



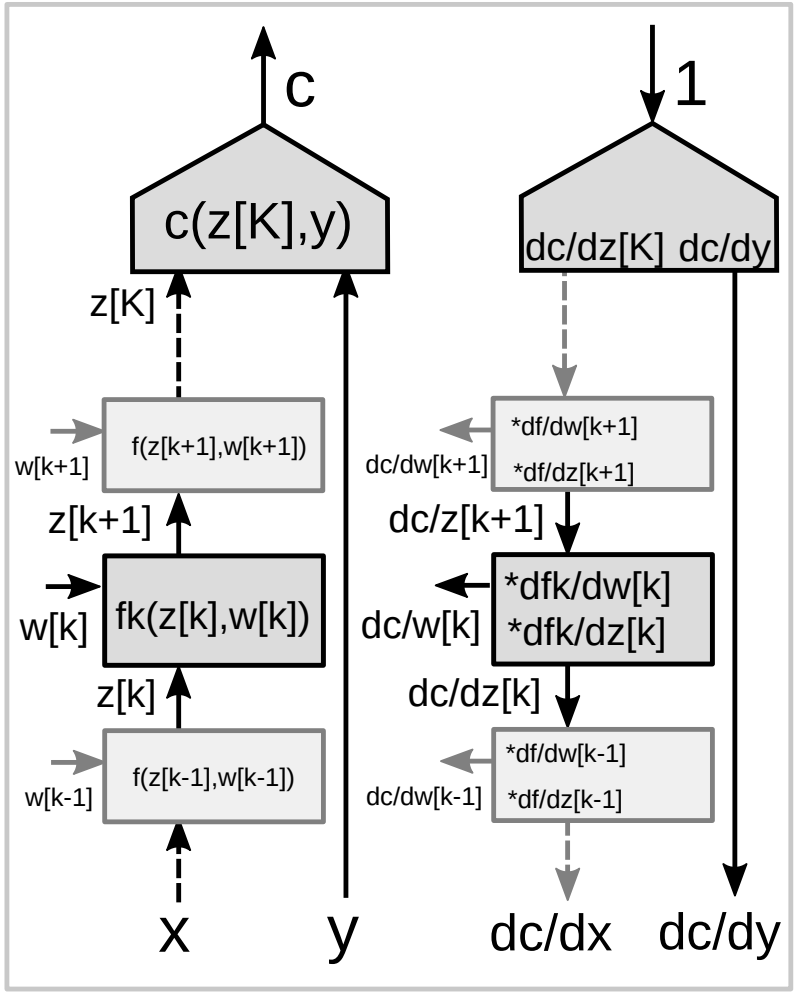
Backprop through a multi-stage graph

- **Using chain rule for vector functions**

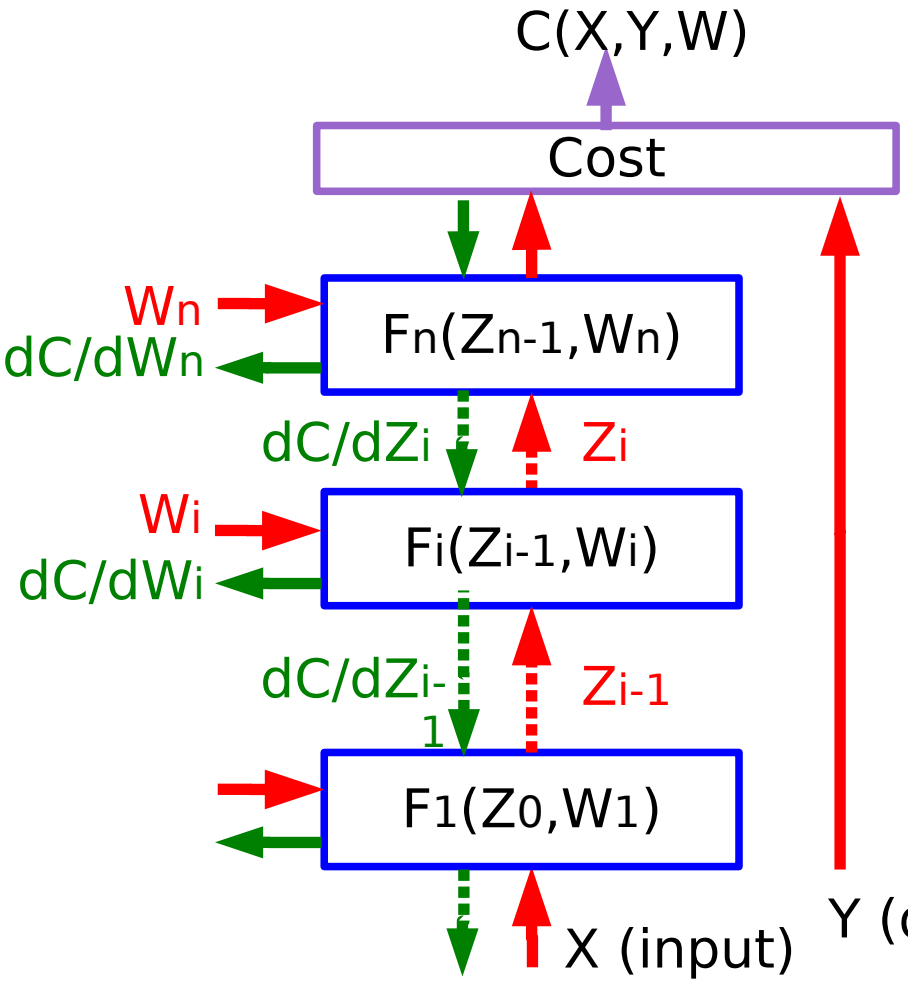
$$\frac{\partial c}{\partial z_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial z_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial f_k(z_k, w_k)}{\partial z_k}$$

$$\frac{\partial c}{\partial w_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial w_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial f_k(z_k, w_k)}{\partial w_k}$$

- **Two Jacobian matrices for the module:**
  - One with respect to z[k]
  - One with respect to w[k]



Computing Gradients by Back-Propagation



- **A practical Application of Chain Rule**

- **Backprop for the state gradients:**
- **$dC/dZ_{i-1} = dC/dZ_i \,.\, dZ_i/dZ_{i-1}$**
- **$dC/dZ_{i-1} = dC/dZ_i \,.\, dF_i(Z_{i-1},W_i)/dZ_{i-1}$**

- **Backprop for the weight gradients:**
- **$dC/dW_i = dC/dZ_i \,.\, dZ_i/dW_i$**
- **$dC/dW_i = dC/dZ_i \,.\, dF_i(Z_{i-1},W_i)/dW_i$**

- **Much more on this later……**

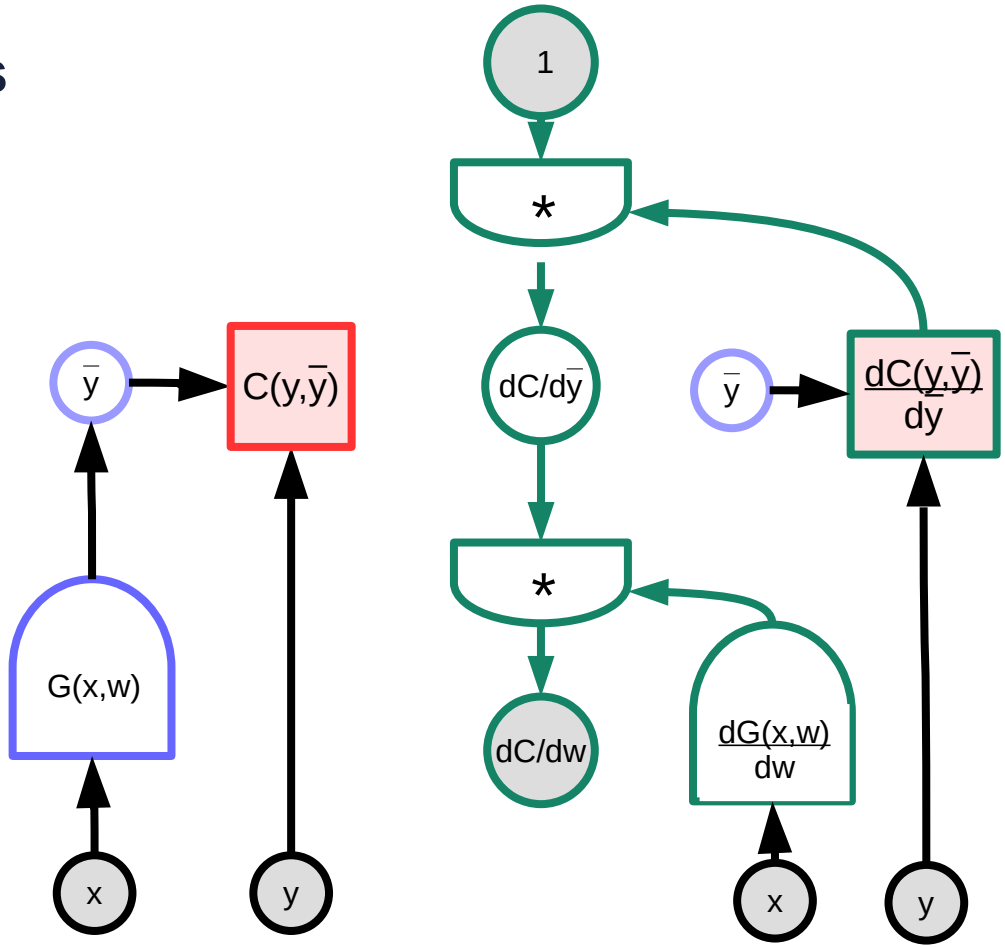
Backprop = propagation through a transformed graph

- **Derivative of composed functions**

$$C(G(w))' = C'(G(w))G'(w)$$

$$\frac{\partial C(y,\bar{y})}{\partial w} = \frac{\partial C(y,\bar{y})}{\boxed{\partial \bar{y}}}\frac{\boxed{\partial \bar{y}}}{\partial w}$$

$$\frac{\partial C(y,\bar{y})}{\partial w} = \frac{\partial C(y,\bar{y})}{\partial \bar{y}}\frac{\partial G(x,w)}{\partial w}$$



Gradient, Jacobian, ….

- **Dimensions:**

$$y, \bar{y} : [M \times 1] \quad w : [N \times 1]$$

$$\frac{\partial C(y,\bar{y})}{\partial w} = \frac{\partial C(y,\bar{y})}{\partial \bar{y}} \frac{\partial \bar{y}}{\partial w}$$

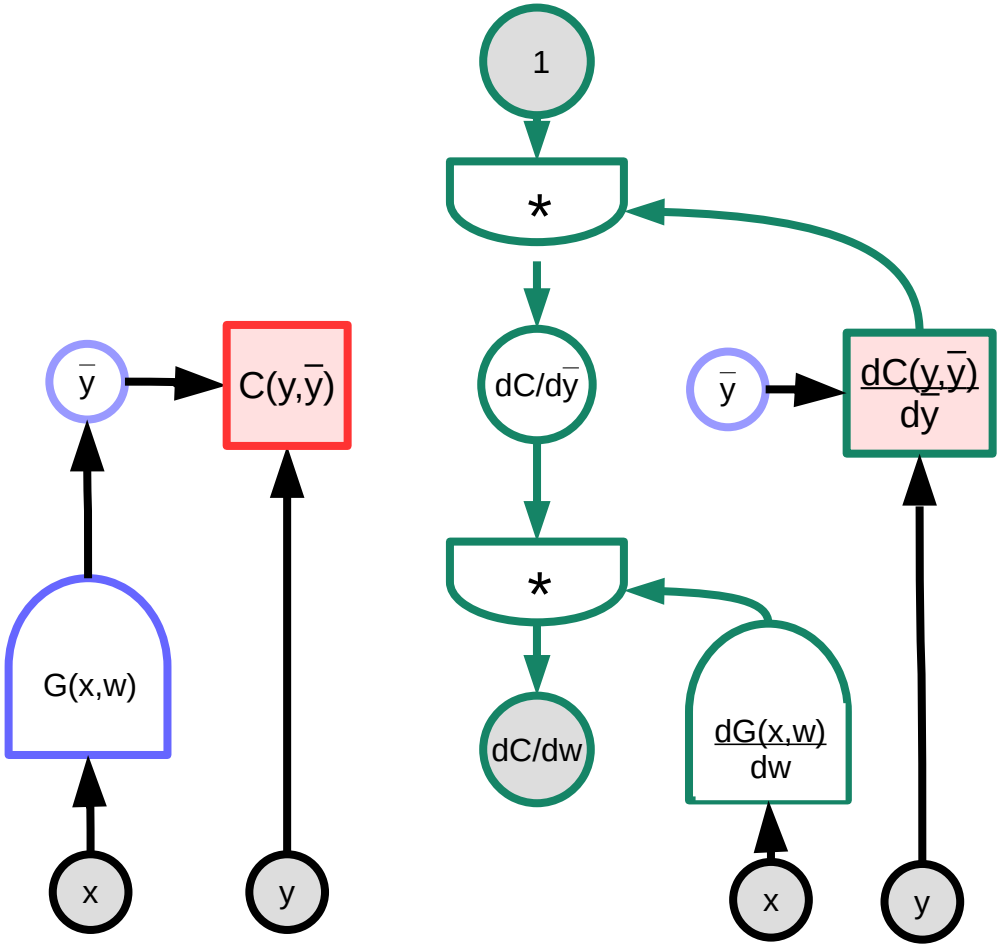
$$[1 \times N] = [1 \times M] \cdot [M \times N]$$

- **Row vector = row vector . matrix**

$$\frac{\partial C(y,\bar{y})}{\partial w} = \frac{\partial C(y,\bar{y})}{\partial \bar{y}} \frac{\partial G(x,w)}{\partial w}$$

$$[1 \times N] = [1 \times M] \cdot [M \times N]$$

- **Gradient = gradient . Jacobian**



Basic Modules

Linear	$Y = W.X$; $dC/dX = W^T.\ dC/dY$; $dC/dW = X\ dC/dY$
ReLU	$y = ReLU(x)$; if $(x<0)$ $dC/dx = 0$ else $dC/dx = dC/dy$
Duplicate	$Y1 = X, Y2 = X$; $dC/dX = dC/dY1 + dC/dY2$
Add	$Y = X1 + X2$; $dC/dX1 = dC/dY$; $dC/dX2 = dC/dY$
Max	$y = \max(x1,x2)$; if $(x1>x2)$ $dC/dx1 = dC/dy$ else $dC/dx1=0$
LogSoftMax	$Yi = Xi - \log[\sum_j \exp(Xj)]$; .....???

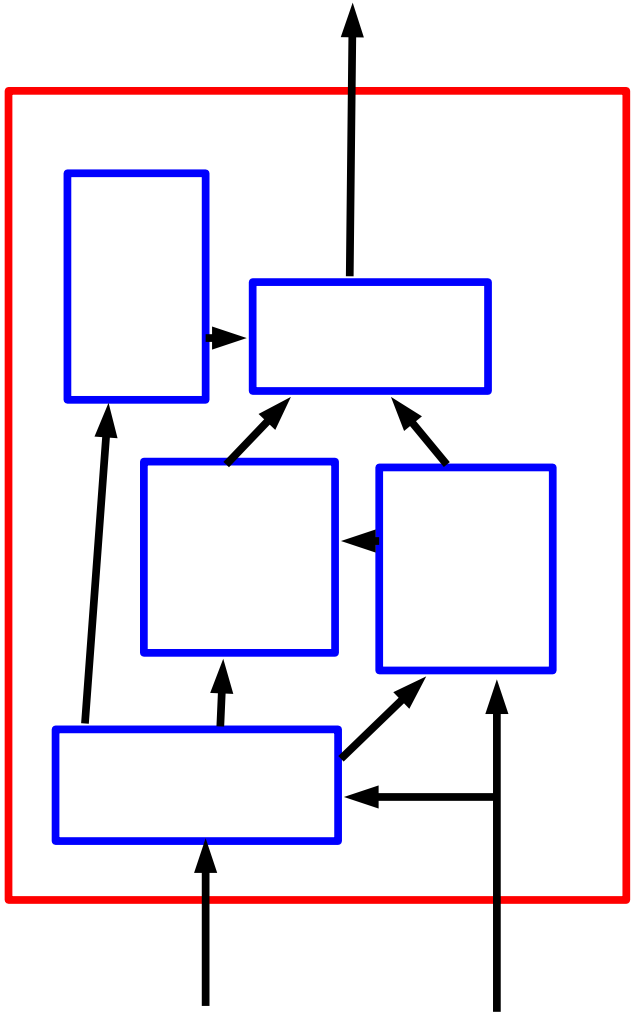
Non-Linear functions and Loss functions in PyTorch

- **ReLu, sigmoids and variations**
- **Squared error, cross-entropy, hinge, ranking loss and variants**

Any directed acyclic graph is OK for backprop

- **As long as there exist a partial order on the modules**

- **If the graph has loops, we need to "unroll" them.**
  - Recurrent networks and backprop through time



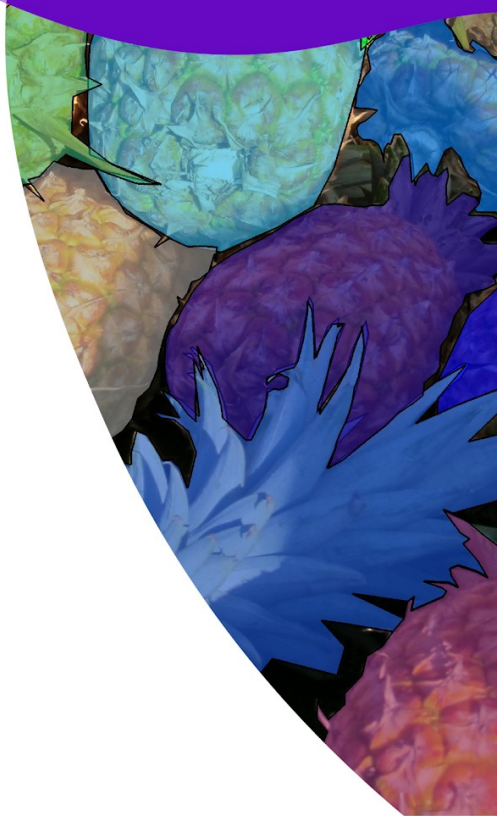
Backprop in Practice

- **Use ReLU non-linearities (tanh and logistic are falling out of favor)**
- **Initialize the weights properly**
- **Use cross-entropy loss for classification**
- **Use Stochastic Gradient Descent on minibatches**
- **Shuffle the training samples**
- **Normalize the input variables (zero mean, unit variance)**
- **Schedule to decrease the learning rate**
- **Use a bit of L1 or L2 regularization on the weights (or a combination)**
  - But it's best to turn it on after a couple of epochs
- **Use “dropout” for regularization**
  - Hinton et al 2012 http://arxiv.org/abs/1207.0580
- **Lots more in [LeCun et al. “Efficient Backprop” 1998]**
- **Lots, lots more in recent papers.**

NEW YORK UNIVERSITY

Learning Representations

What are good representations?
Why do networks need to be deep?



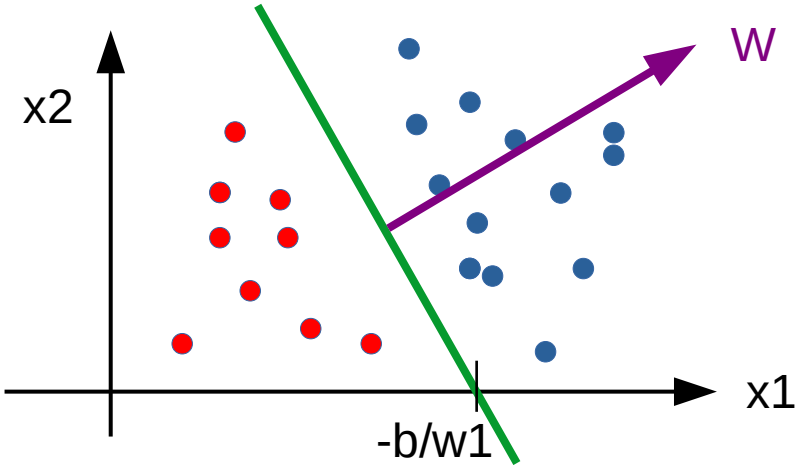
Linear Classifiers and their limitations

- **Linear classifier**

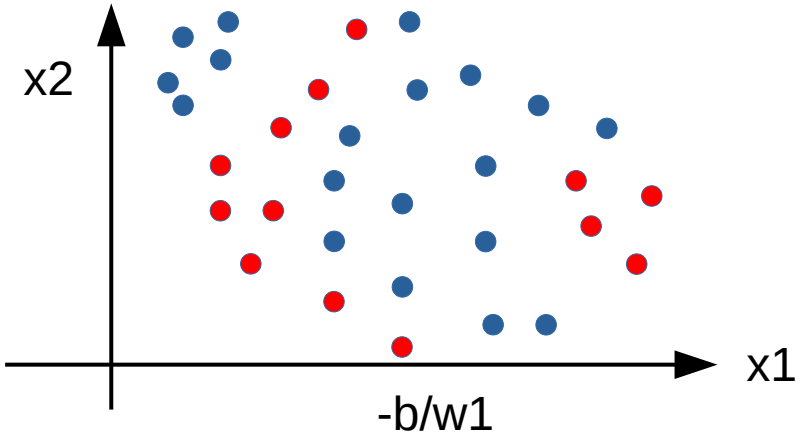
$$\bar{y} = sign\left(\sum_{i=1}^{N} w_i x_i + b\right)$$

- Partitions the space into two half spaces separated by the hyperplane:

$$\sum_{i=1}^{N} w_i x_i + b = 0$$

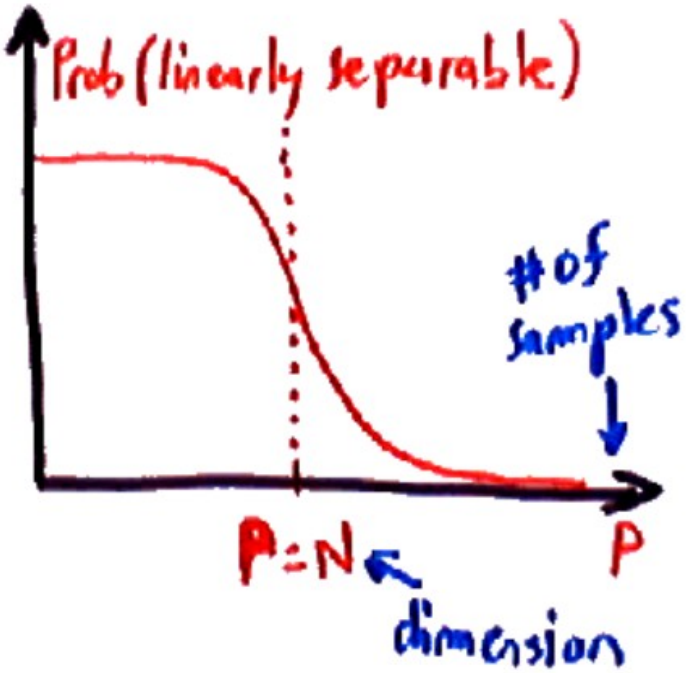


Not linearly separable dataset



Number of linearly separable dichotomies

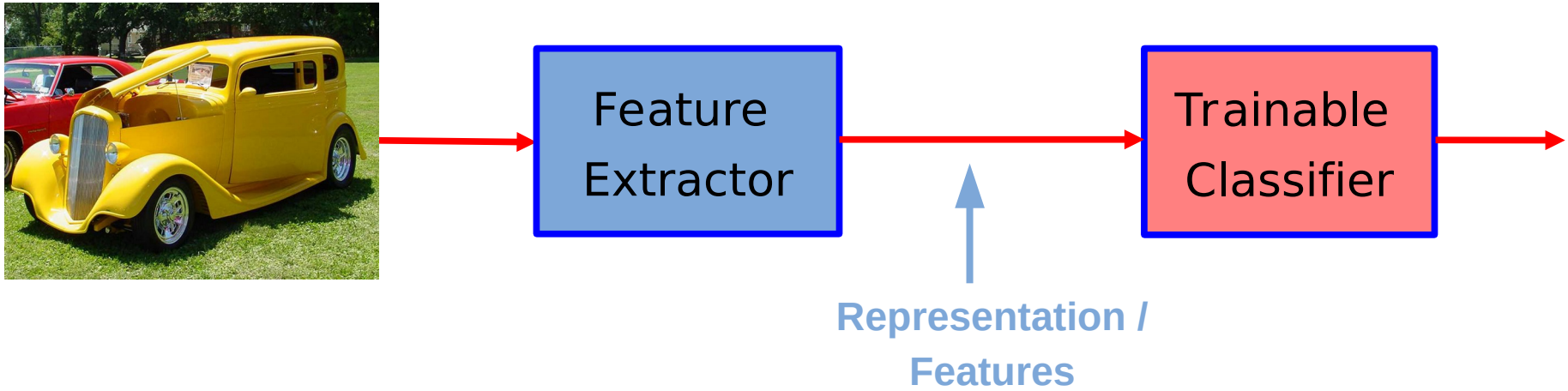
- **The probability that a dichotomy over P points in N dimensions is linearly separable goes to zero as P gets larger than N**
  - [Cover's theorem 1966]



- Problem: there are 2^P possible dichotomies of P points.
- Only about N are linearly separable.
- If P is larger than N, the probability that a random dichotomy is linearly separable is very, very small.

Solution: representations (a.k.a. features)

- **Extracting relevant features from the raw input**
- **Computing good representations of the input**
- **The feature extractor <u>must</u> be non-linear**
- **Simple solution: expand the dimension non-linearly**
  - But how?



Ideas for “generic” feature extraction

- **Basic principle:**
  - expanding the dimension of the representation so that things are more likely to become linearly separable.

- **- space tiling**
- **- random projections**
- **- polynomial classifier (feature cross-products)**
- **- radial basis functions**
- **- kernel machines**

Example: monomial features

- **Feature extractor computes cross products of input variables**
- **A linear classifier on top computes a polynomial of input variables**

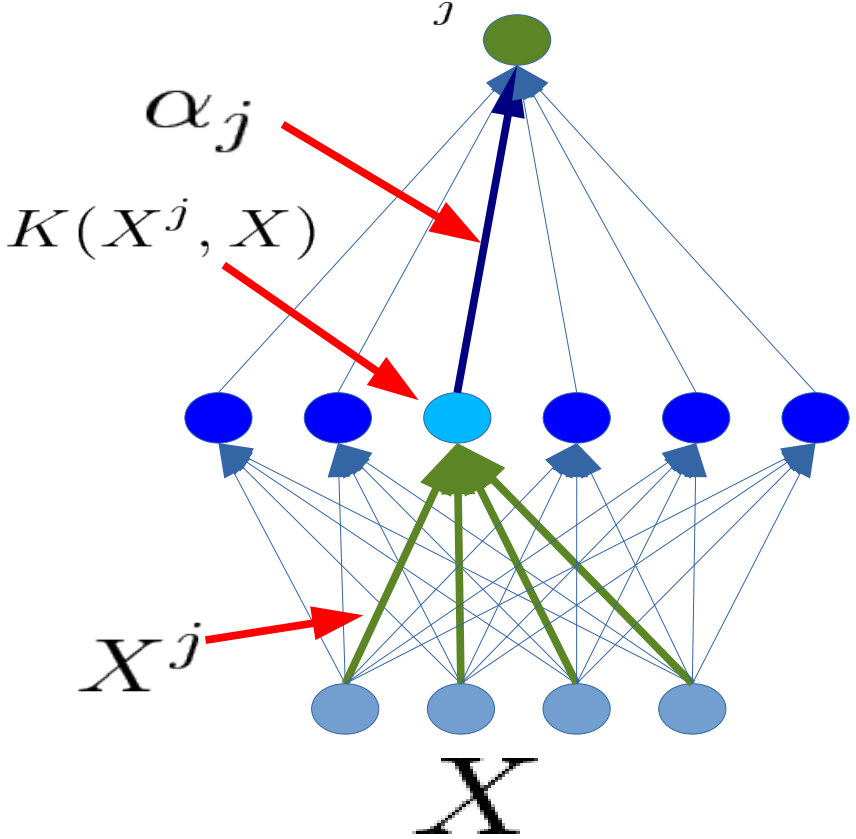
$$\Phi(x_1, x_2) = [1, x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

- **generalizable to degree d**
- **Unfortunately impractical for large d**
- **Number of features is d choose N, which grows like N^d**
- **But d=2 is used a lot in "attention" circuits.**

Shallow networks are universal approximators!

- **SVMs and Kernel methods**
  - Layer1: kernels; layer2: linear
  - The first layer is “trained” with the simplest unsupervised method ever devised: using the samples as templates for the kernel functions.
- **2-layer neural nets**
  - Layer1: dot products + non-linear function; Layer2: linear
- **But few useful functions can be efficiently represented with only two layers of reasonable size.**

$$G(X,\alpha) = \sum_j \alpha_j K(X^j, X)$$



Do we really need deep architectures?

- **Theoretician's dilemma: "We can approximate any function as close as we want with shallow architecture. Why would we need deep ones?"**

$$y = \sum_{i=1}^{P} \alpha_i K(X, X^i) \qquad y = F(W^1.F(W^0.X))$$

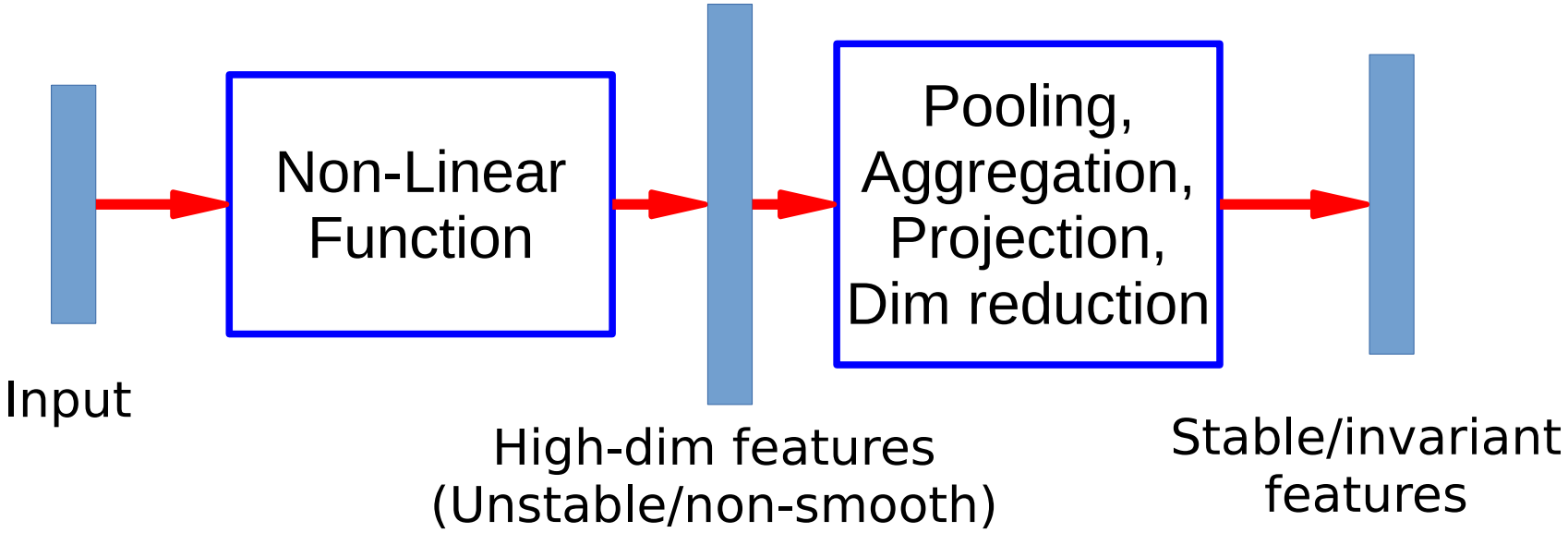
  - kernel machines (and 2-layer neural nets) are "universal".
- **Deep learning machines**

$$y = F(W^K.F(W^{K-1}.F(....F(W^0.X)...)))$$

- **Deep machines are more efficient for representing certain classes of functions, particularly those involved in visual recognition**
  - they can represent more complex functions with less "hardware"
- **We need an efficient parameterization of the class of functions that are useful for "AI" tasks (vision, audition, NLP...)**

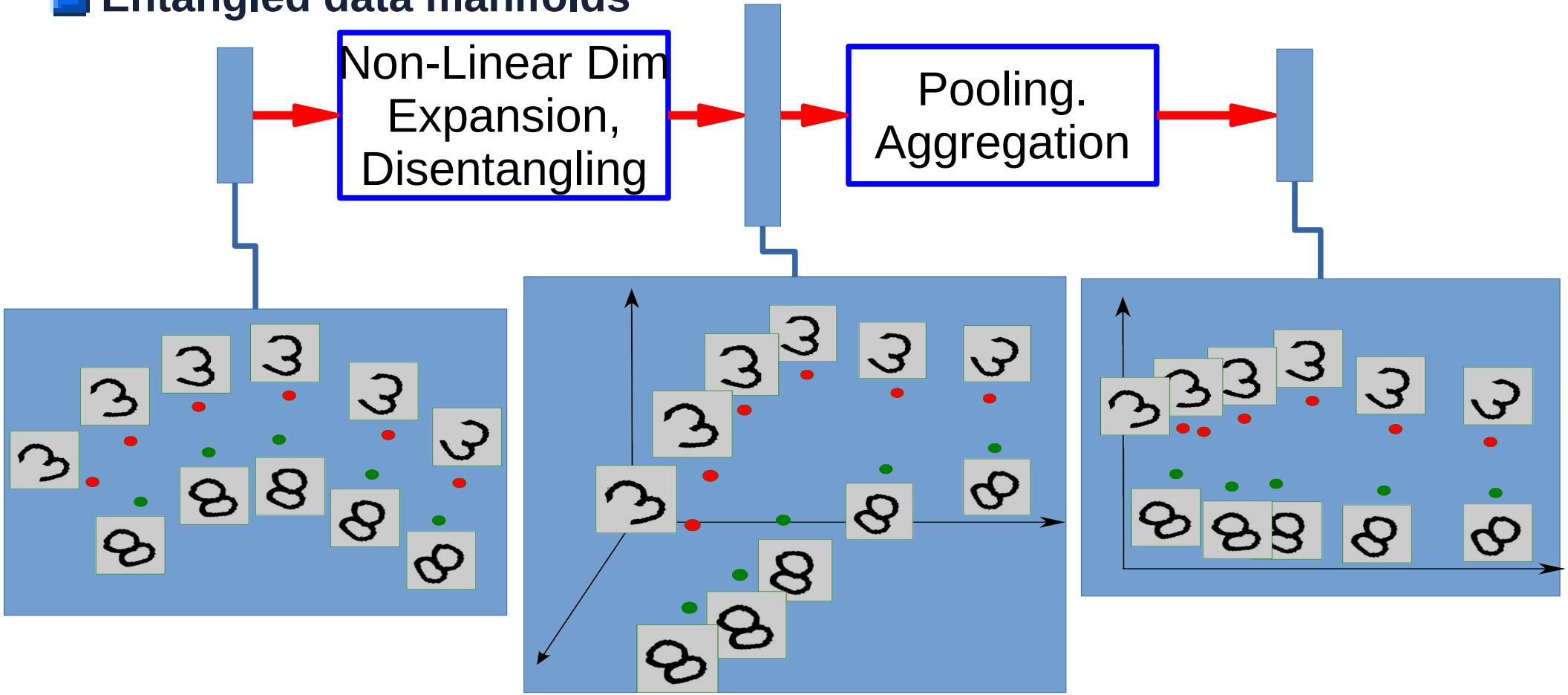
Basic Idea for Invariant Feature Learning

- **Embed the input non-linearly into a high(er) dimensional space**
  - In the new space, things that were non separable may become separable
- **Pool regions of the new space together**
  - Bringing together things that are semantically similar. Like pooling.



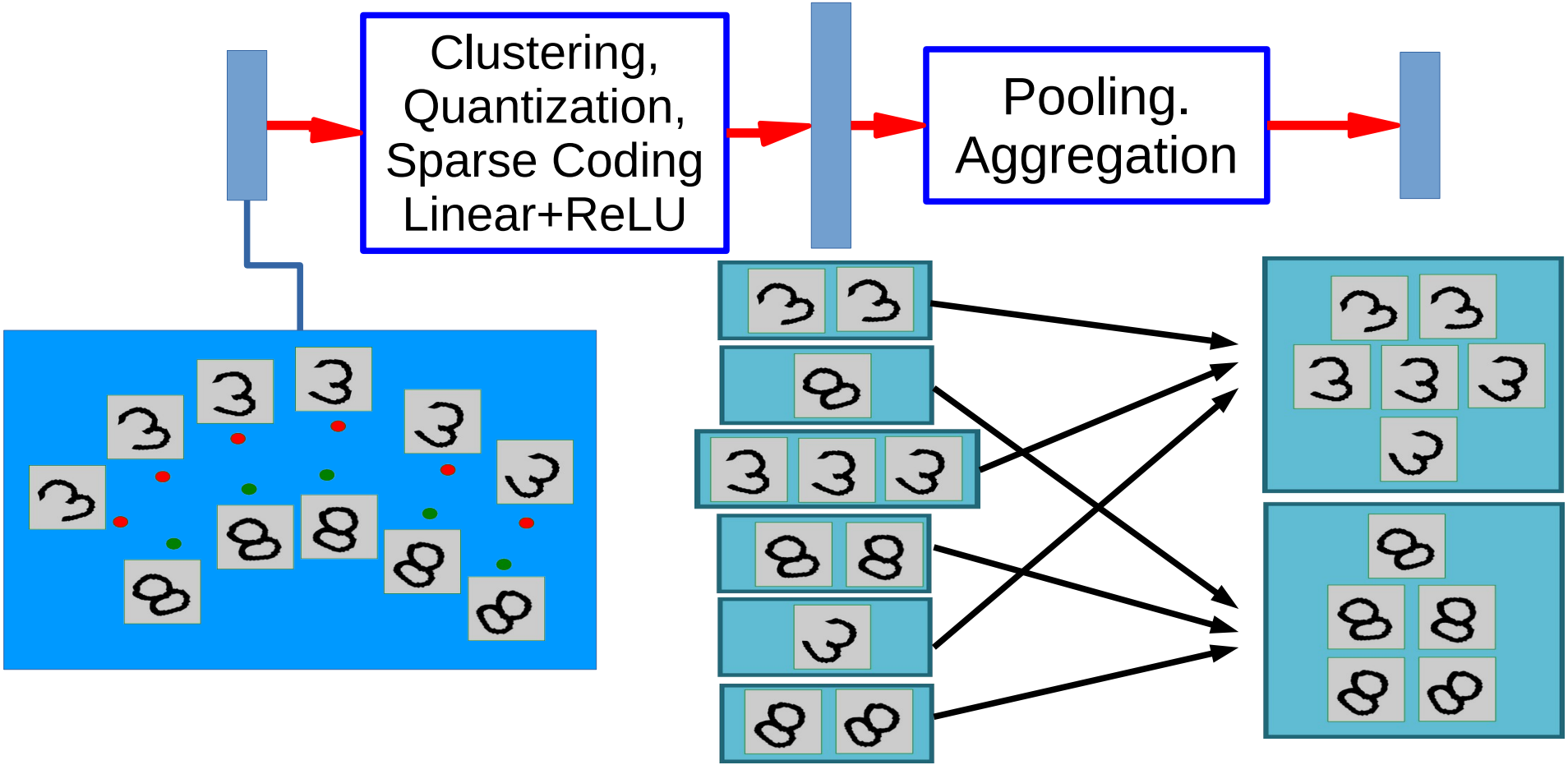
Non-Linear Expansion → Pooling

- **Entangled data manifolds**



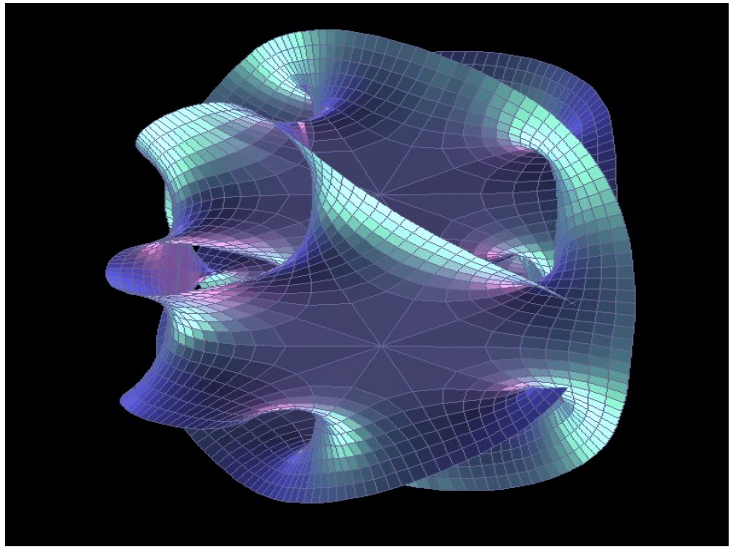
Sparse Non-Linear Expansion → Pooling

- **Use non-linear fn to break things apart, pool together similar things**



Discovering the Hidden Structure in High-Dimensional Data: The manifold hypothesis

- **Learning Representations of Data:**
  - **Discovering & disentangling the independent explanatory factors**
- **The Manifold Hypothesis:**
  - Natural data lives in a low-dimensional (non-linear) manifold
  - Because variables in natural data are mutually dependent



Discovering the Hidden Structure in High-Dimensional Data

- **Example: all face images of a person**
  - 1000x1000 pixels = 1,000,000 dimensions
  - But the face has 3 Cartesian coordinates and 3 Euler angles
  - And humans have less than about 50 muscles in the face
  - Hence the manifold of face images for a person has <56 dimensions
- **The perfect representations of a face image:**
  - Its coordinates on the face manifold
  - Its coordinates away from the manifold
- **We do not have good and general methods to learn functions that turns an image into this kind of representation**

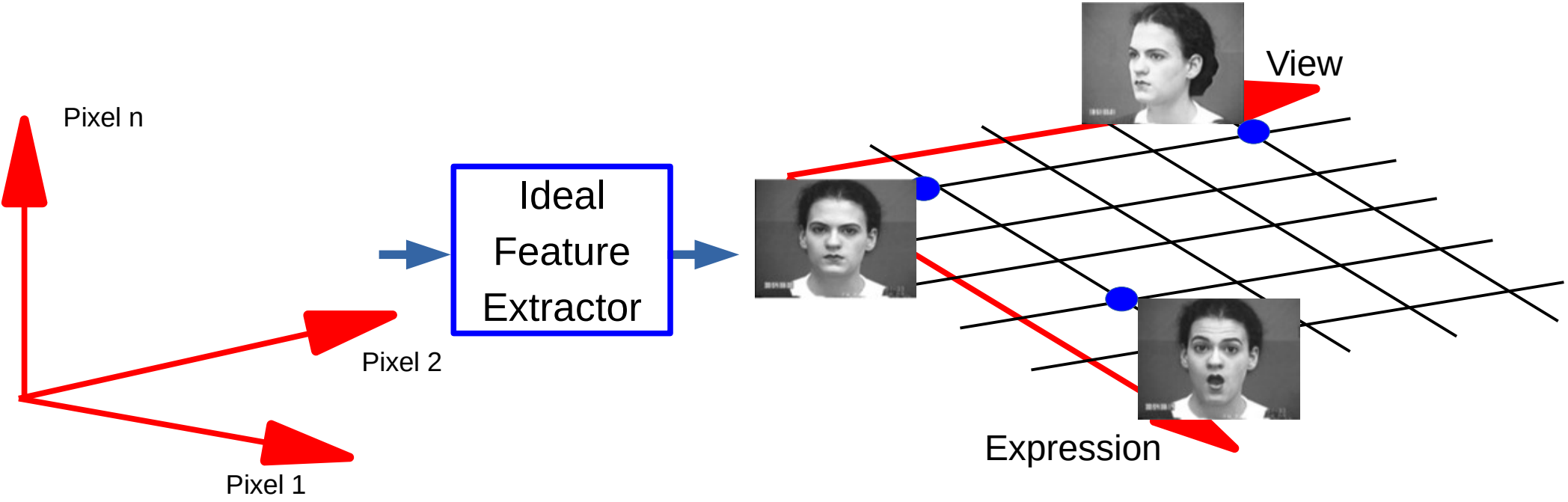
Ideal Feature Extractor

$$\begin{bmatrix} 1.2 \\ -3 \\ 0.2 \\ -2\ldots \end{bmatrix}$$

Face/not face
Pose
Lighting
Expression

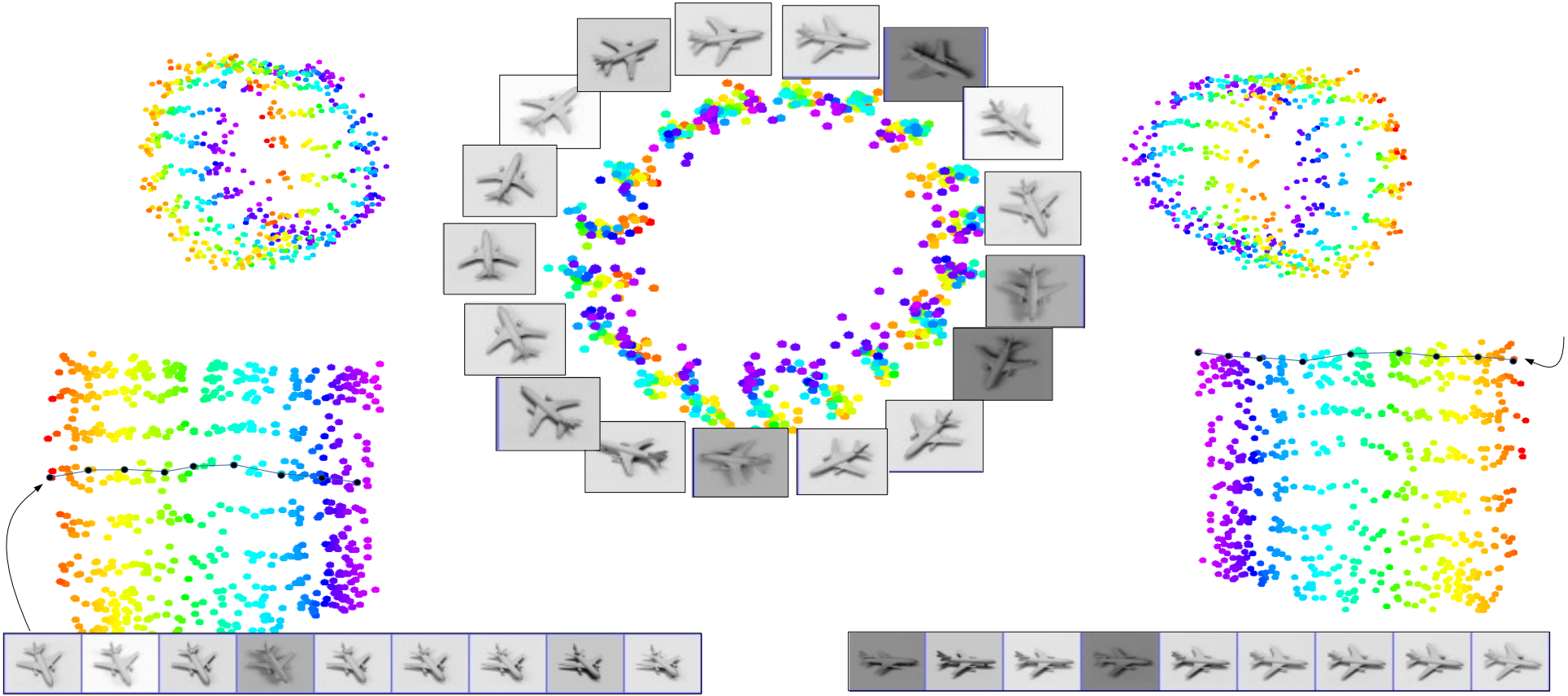
Disentangling factors of variation

The Ideal Disentangling Feature Extractor



Data Manifold

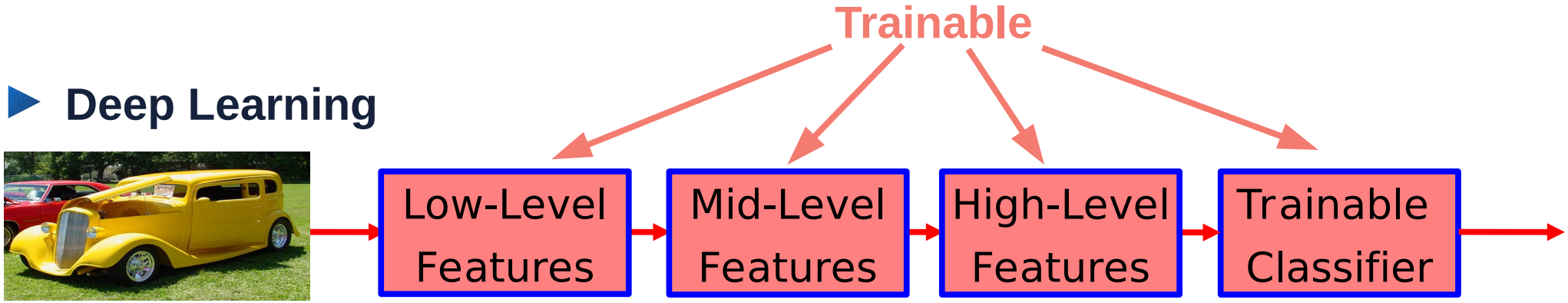
[Hadsell et al. CVPR 2006]



Deep Learning = Learning Hierarchical Representations

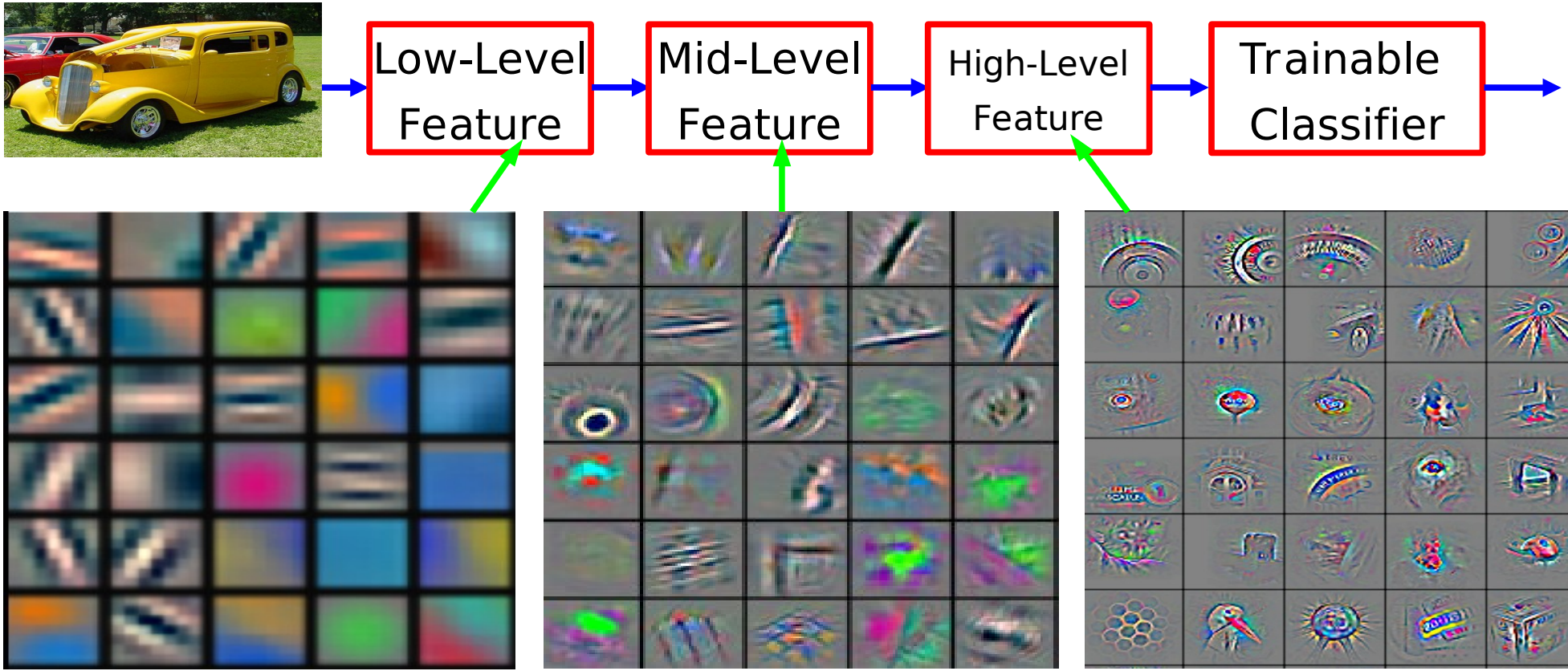
- **Traditional Machine Learning**

- **Deep Learning**



Multilayer Architectures == Compositional Structure of Data

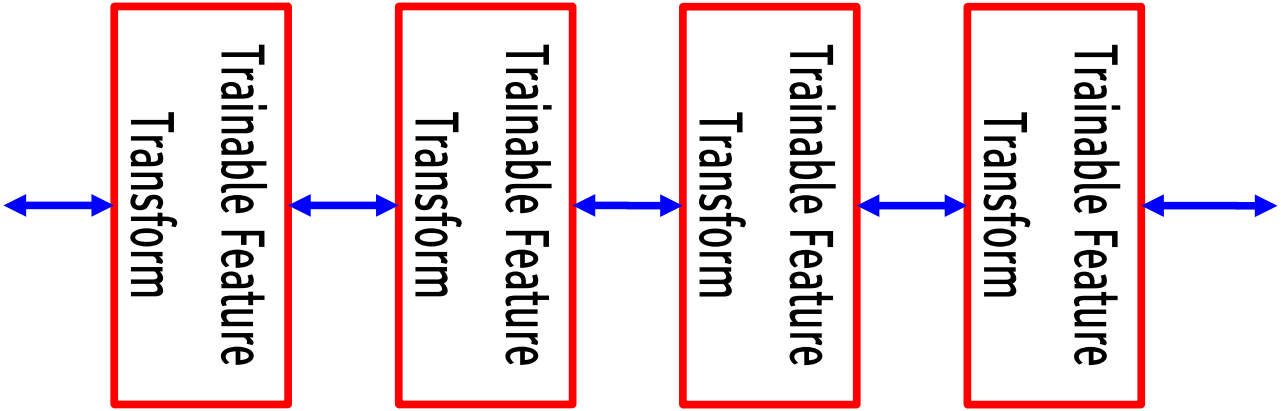
- **Natural is data is compositional => it is efficiently representable hierarchically**



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Multilayer Architecture == Hierarchical representation

- **Hierarchy of representations with increasing level of abstraction**
- **Each stage is a kind of trainable feature transform**
- **Image recognition**
  - Pixel → edge → texton → motif → part → object
- **Text**
  - Character → word → word group → clause → sentence → story
- **Speech**
  - Sample → spectral band → sound → … → phone → phoneme → word



Why would deep architectures be more efficient?

[Bengio & LeCun 2007 “Scaling Learning Algorithms Towards AI”]

- **A deep architecture trades space for time (or breadth for depth)**
  - more layers (more sequential computation),
  - but less hardware (less parallel computation).
- **Example1: N-bit parity**
  - requires N-1 XOR gates in a tree of depth log(N).
  - Even easier if we use threshold gates
  - requires an exponential number of gates of we restrict ourselves to 2 layers (DNF formula with exponential number of minterms).
- **Example2: circuit for addition of 2 N-bit binary numbers**
  - Requires O(N) gates, and O(N) layers using N one-bit adders with ripple carry propagation.
  - Requires lots of gates (some polynomial in N) if we restrict ourselves to two layers (e.g. Disjunctive Normal Form).
  - Bad news: almost all boolean functions have a DNF formula with an exponential number of minterms O(2^N).....