Introduction to Deep Learning

Lecture 01

Yann Le Cun

Facebook AI Research,

Center for Data Science, NYU

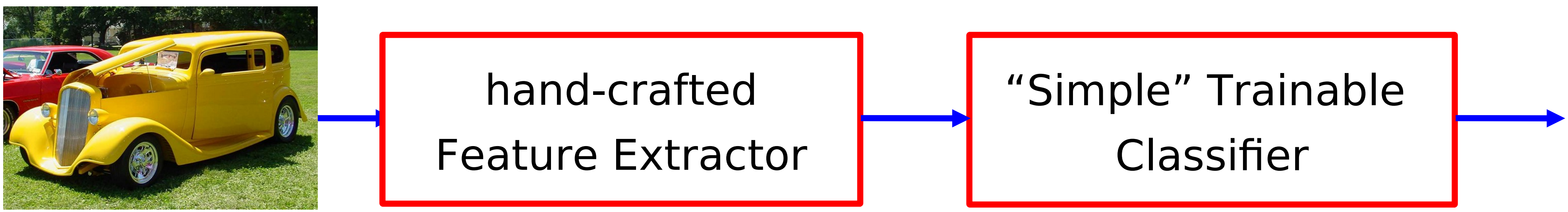
Courant Institute of Mathematical Sciences, NYU

http://yann.lecun.com

Deep Learning = Learning Representations/Features

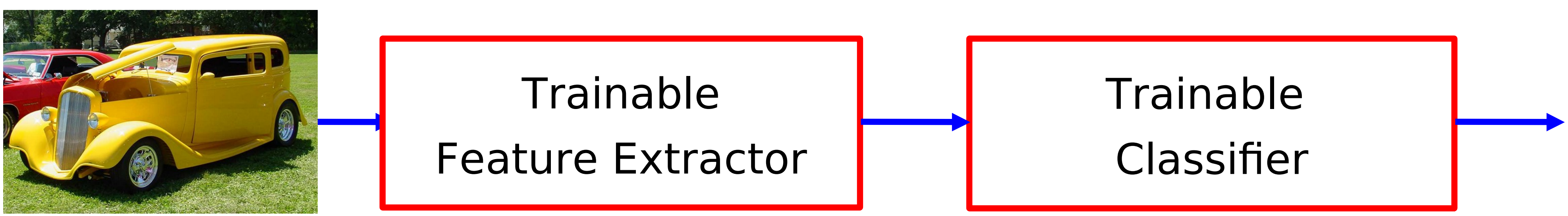
The traditional model of pattern recognition (since the late 50's)

- Fixed/engineered features (or fixed kernel) + trainable classifier



End-to-end learning / Feature learning / Deep learning

- Trainable features (or kernel) + trainable classifier



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

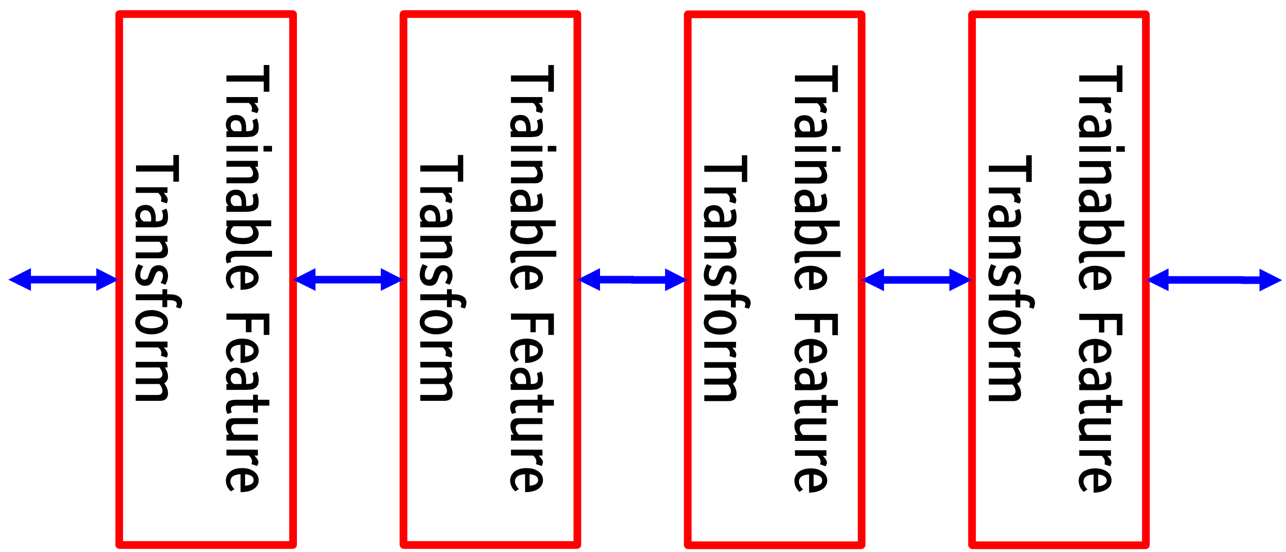
- polynomial classifier (feature cross-products)

- radial basis functions

- kernel machines

Hierarchical representation

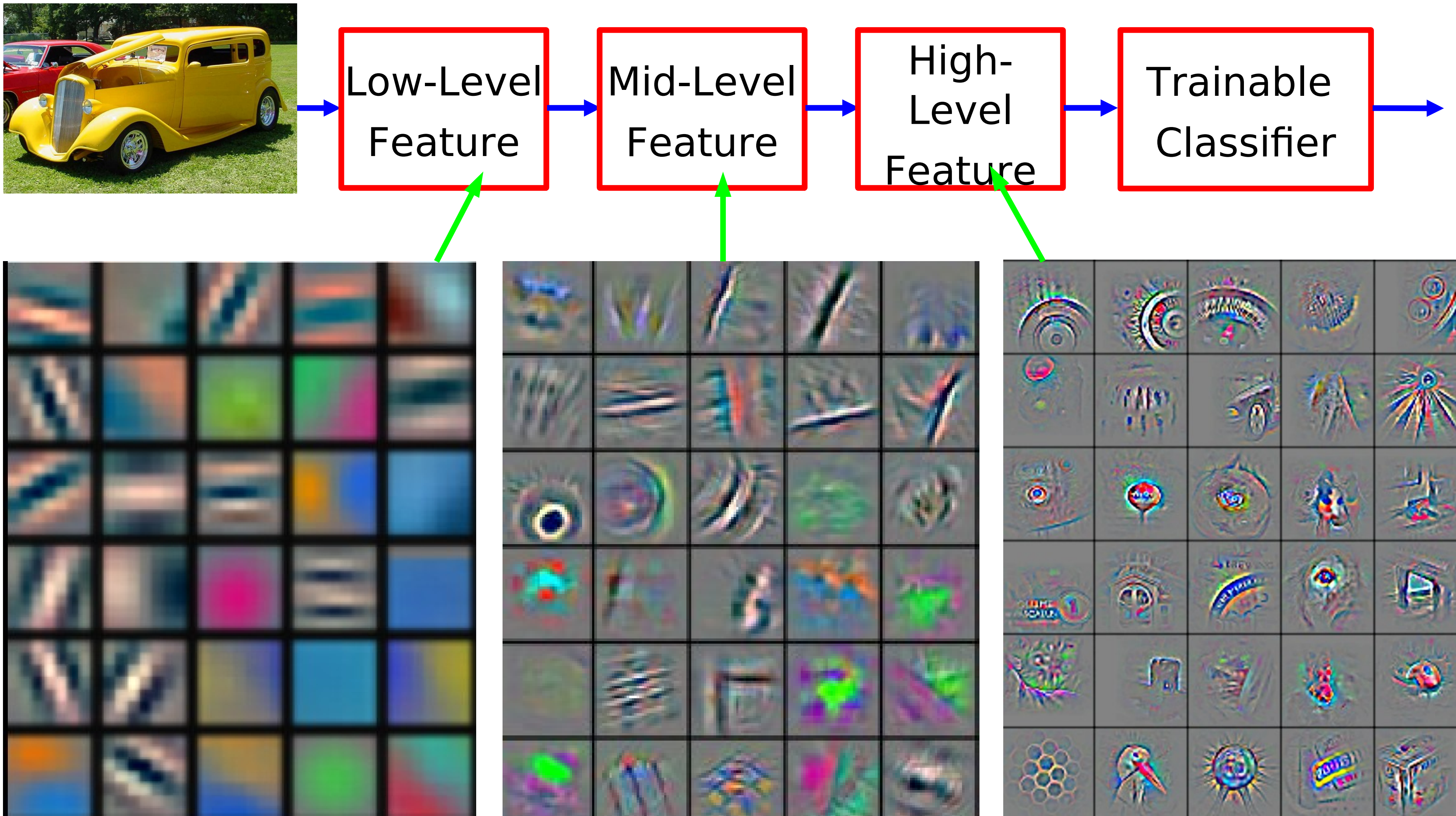
- **Hierarchy of representations with increasing level of abstraction**
- **Each stage is a kind of trainable feature transform**
- **Image recognition**
  - Pixel → edge → texton → motif → part → object
- **Text**
  - Character → word → word group → clause → sentence → story
- **Speech**
  - Sample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



Deep Learning = Learning Hierarchical Representations

Y LeCun
MA Ranzato

- It's deep if it has more than one stage of non-linear feature transformation

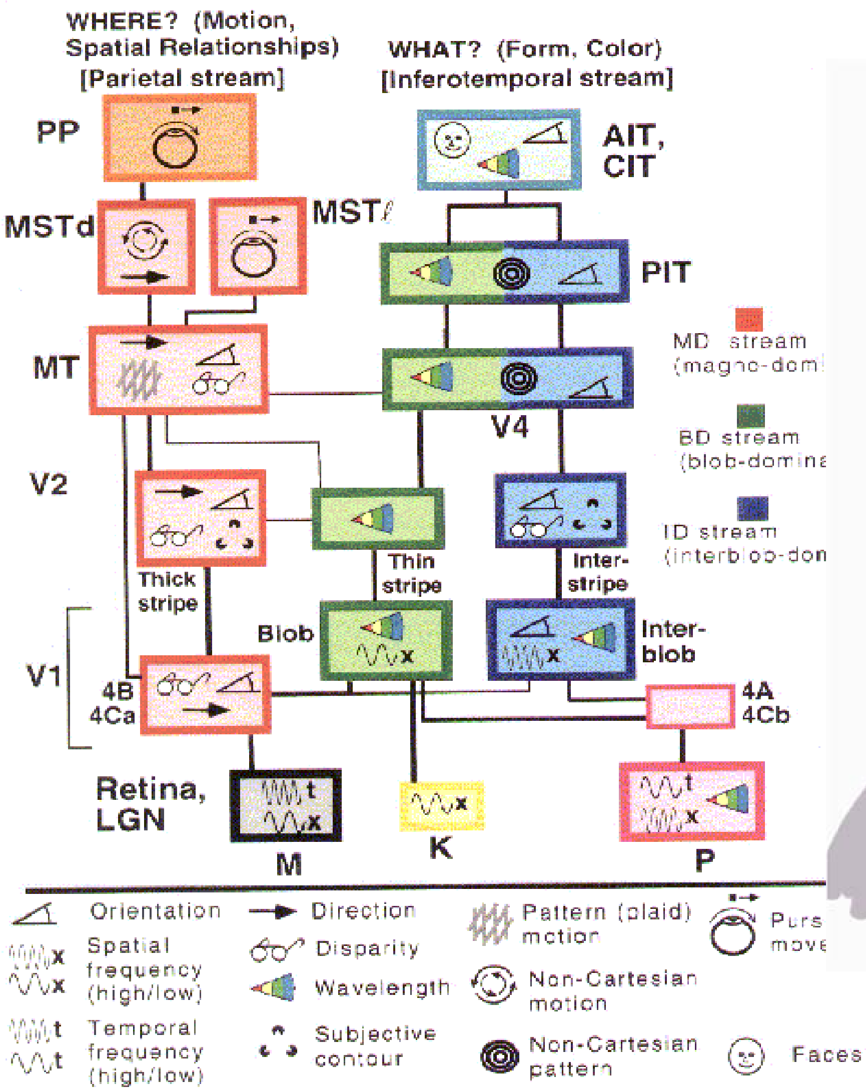


Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

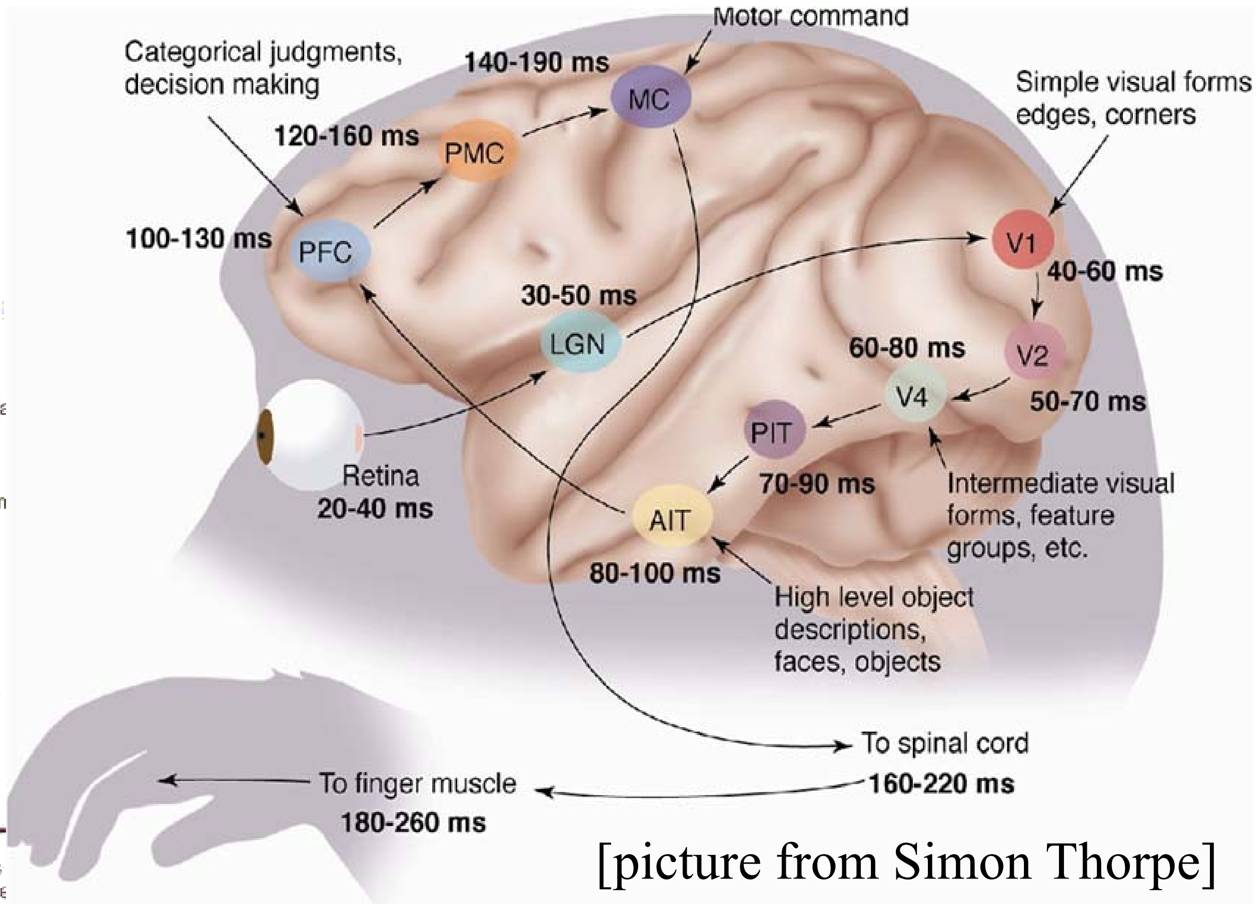
The Mammalian Visual Cortex is Hierarchical

Y LeCun
MA Ranzato

- The ventral (recognition) pathway in the visual cortex has multiple stages
- Retina - LGN - V1 - V2 - V4 - PIT - AIT ....



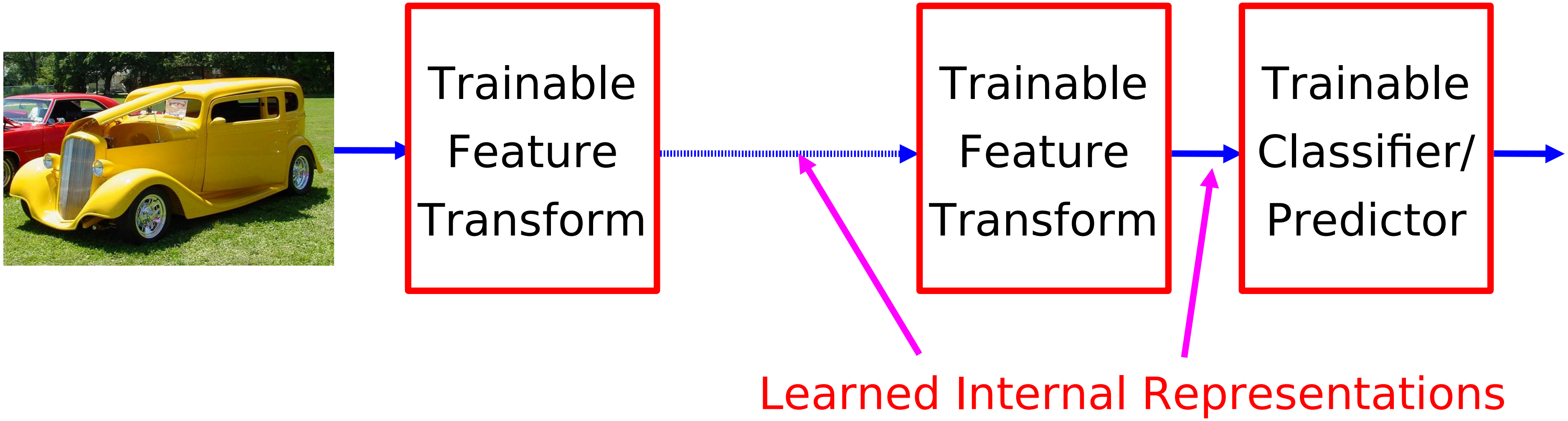
[Gallant & Van Essen]



[picture from Simon Thorpe]

A hierarchy of trainable feature transforms

- Each module transforms its input representation into a higher-level one.
- High-level features are more global and more invariant
- Low-level features are shared among categories



How can we make all the modules trainable and get them to learn appropriate representations?

Do we really need deep architectures?

Y LeCun
MA Ranzato

- **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...)**

Why would deep architectures be more efficient?

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

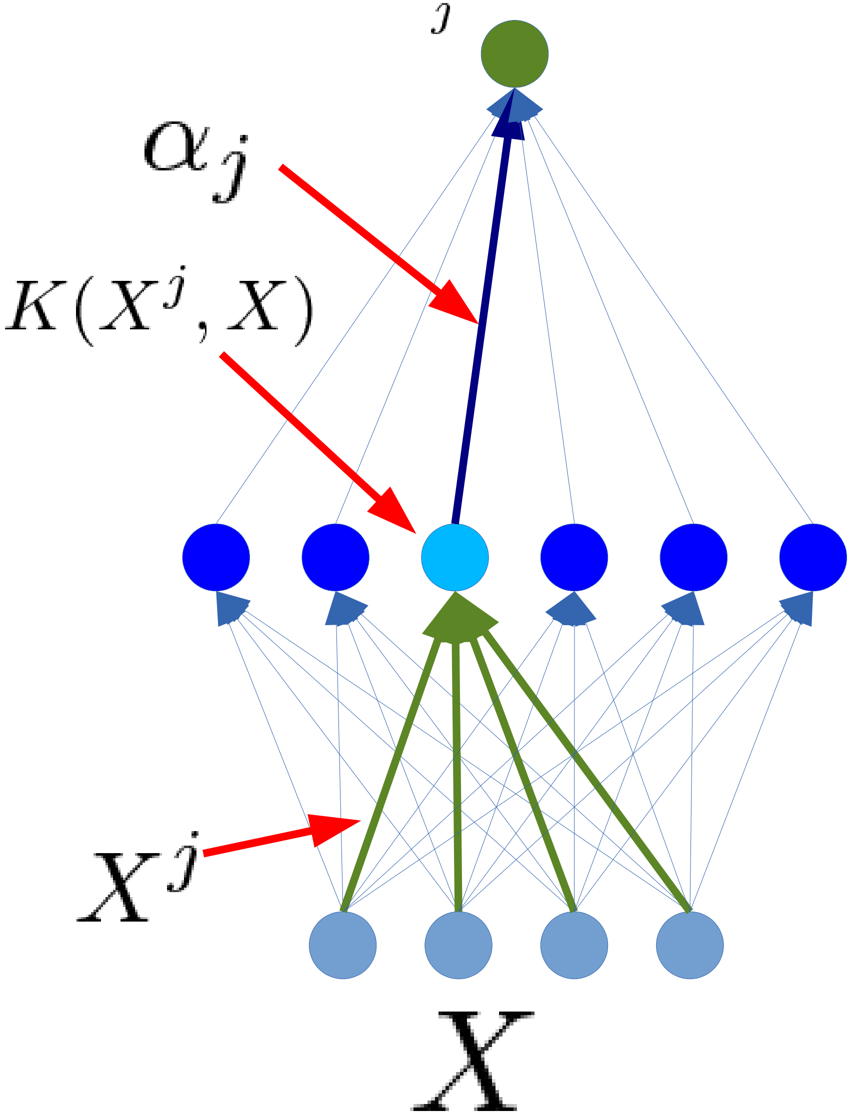
- **A deep architecture trades space for time (or breadth for depth)**
  - more layers (more sequential computation),
  - but less hardware (less parallel computation).
- **Example1: N-bit parity**
  - requires N-1 XOR gates in a tree of depth log(N).
  - Even easier if we use threshold gates
  - requires 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).....

Which Models are Deep?

Y LeCun
MA Ranzato

- **2-layer models are not deep (even if you train the first layer)**
  - Because there is no feature hierarchy
- **Neural nets with 1 hidden layer are not deep**
- **SVMs and Kernel methods are not deep**
  - Layer1: kernels; layer2: linear
  - The first layer is “trained” in with the simplest unsupervised method ever devised: using the samples as templates for the kernel functions.
- **Classification trees are not deep**
  - No hierarchy of features. All decisions are made in the input space

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

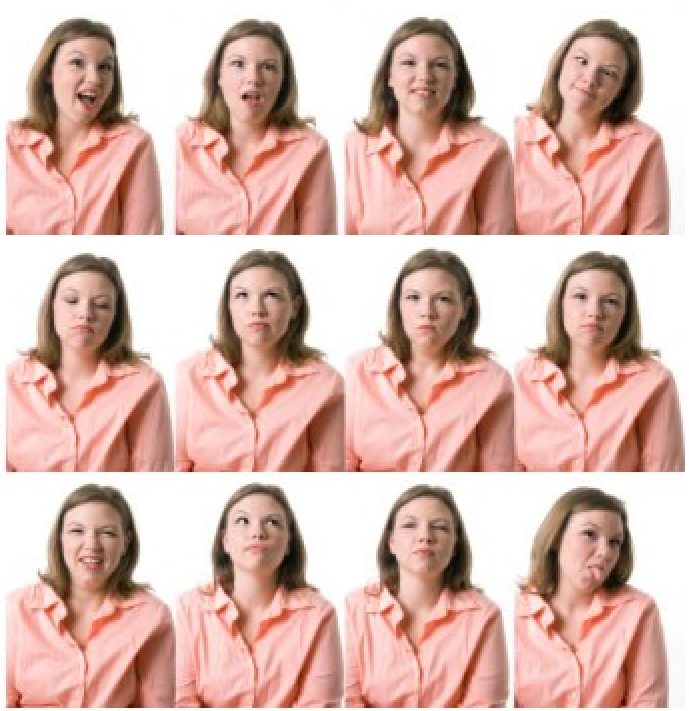


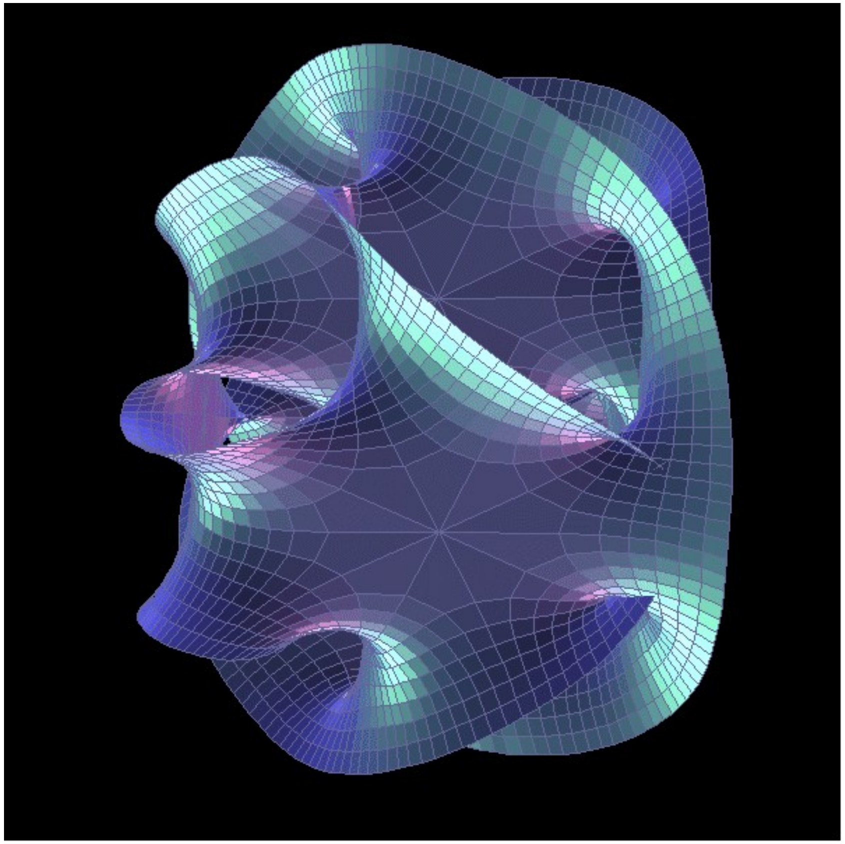
What Are
Good Feature?

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

Y LeCun
MA Ranzato

- **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





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**

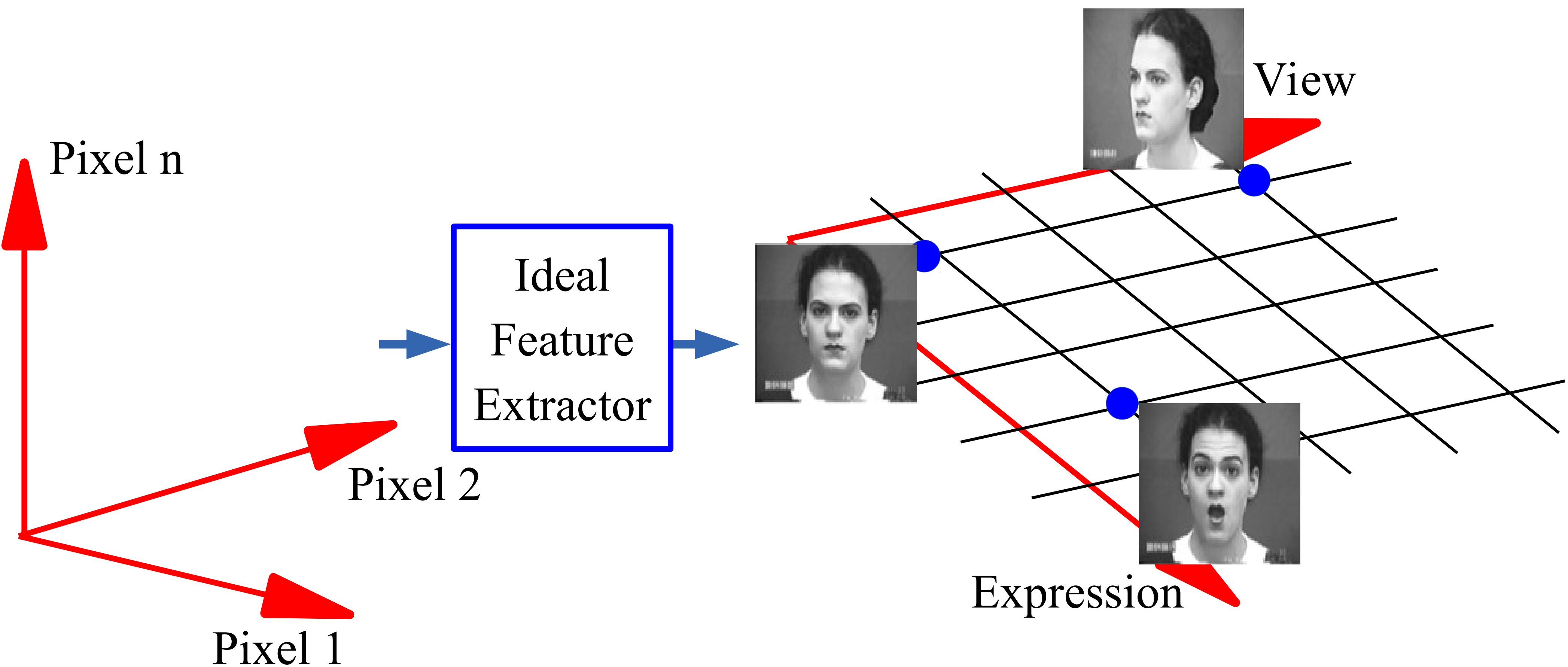
$\rightarrow$ Ideal Feature Extractor $\rightarrow$

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

Face/not face
Pose
Lighting
Expression

Disentangling factors of variation

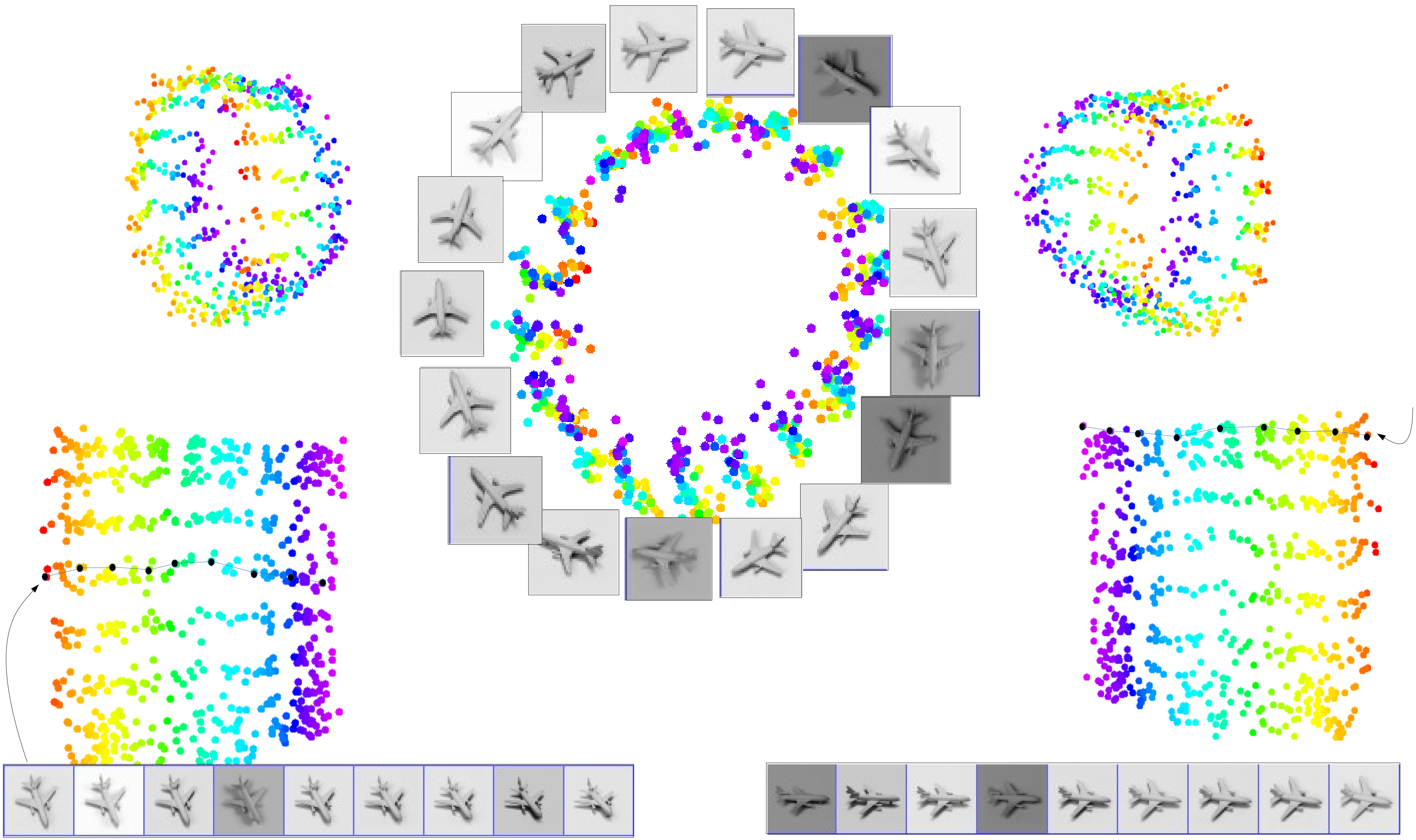
The Ideal Disentangling Feature Extractor



Y LeCun
MA Ranzato

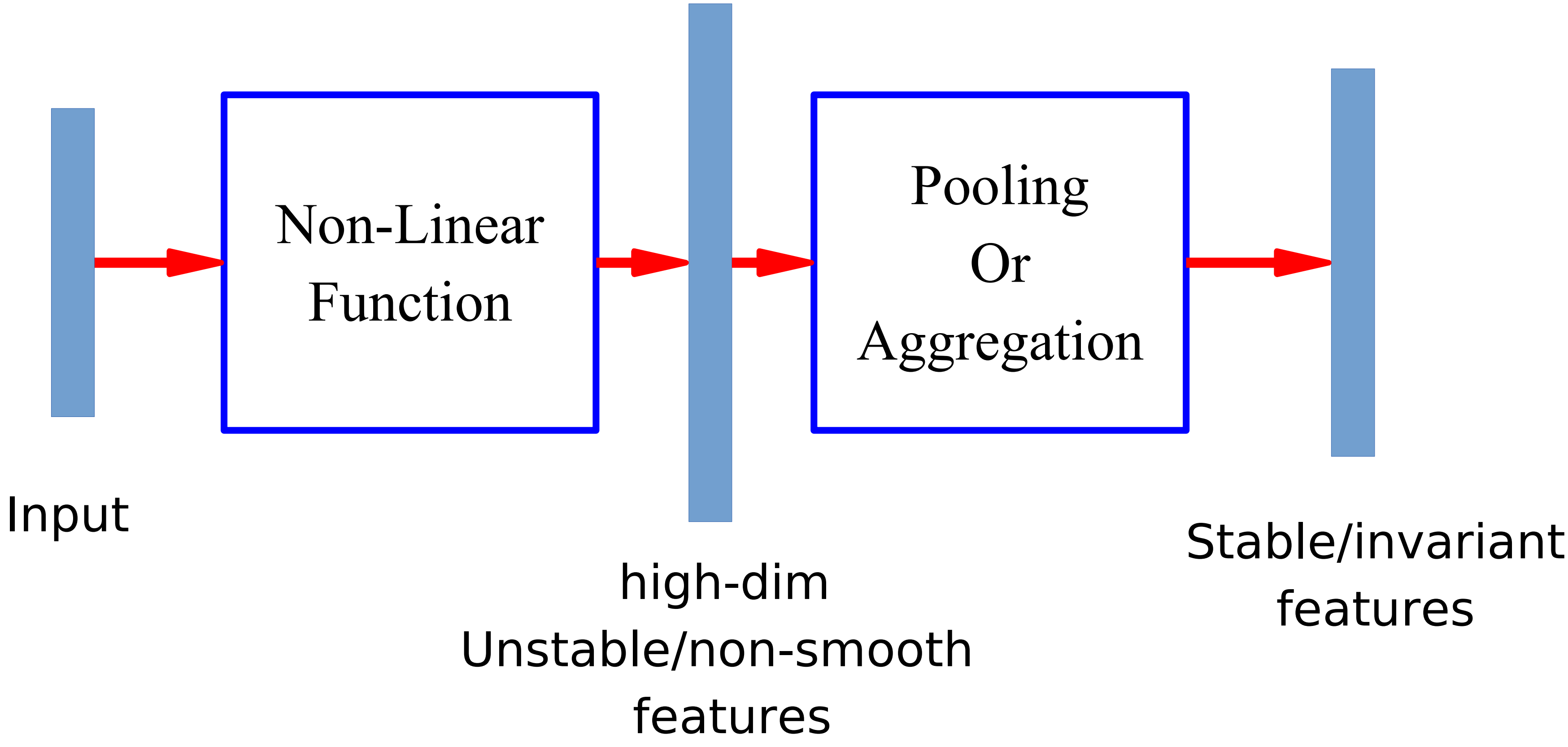
[Hadsell et al. CVPR 2006]

- **Azimuth-Elevation manifold. Ignores lighting.**



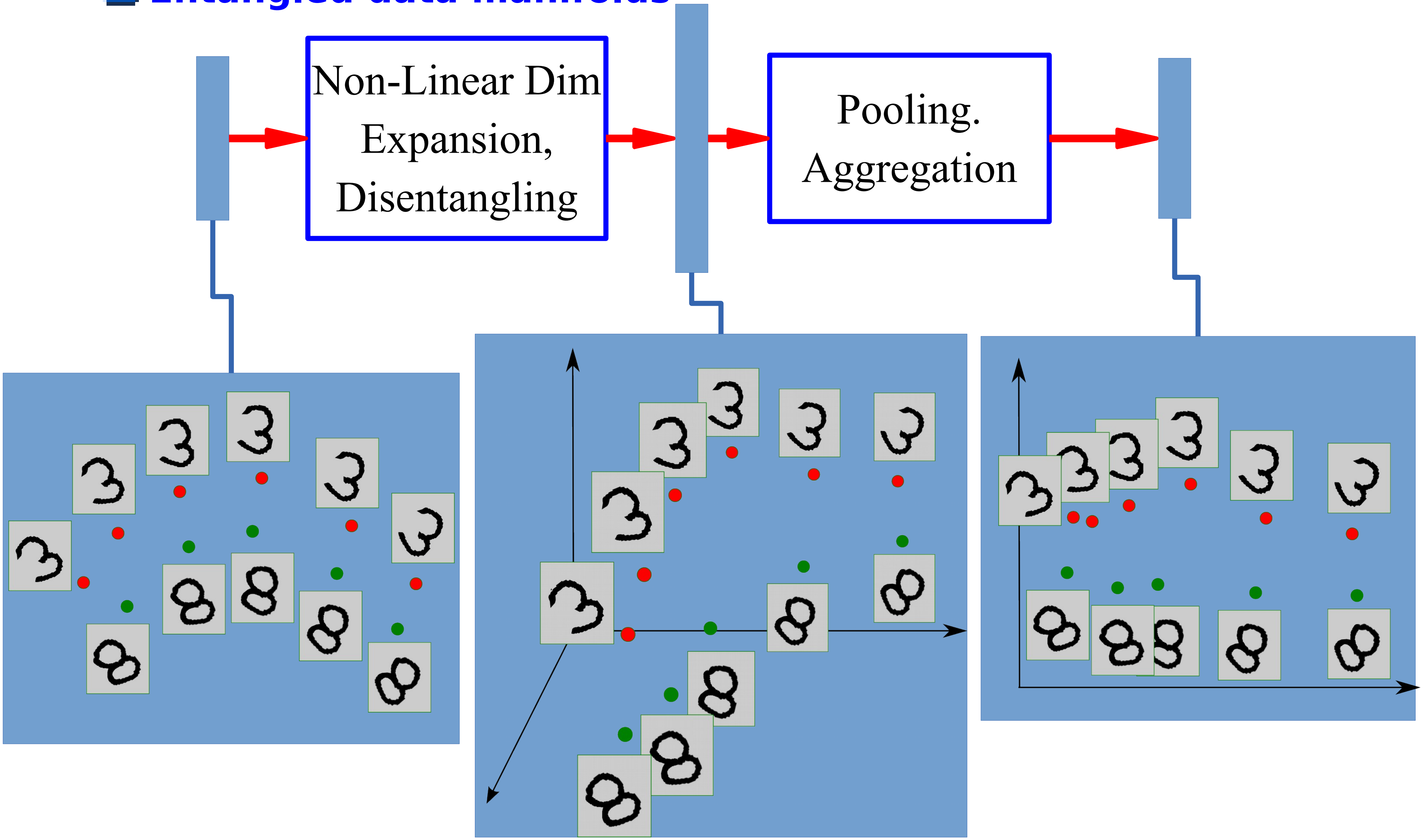
Basic Idea for Invariant Feature Learning

- **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

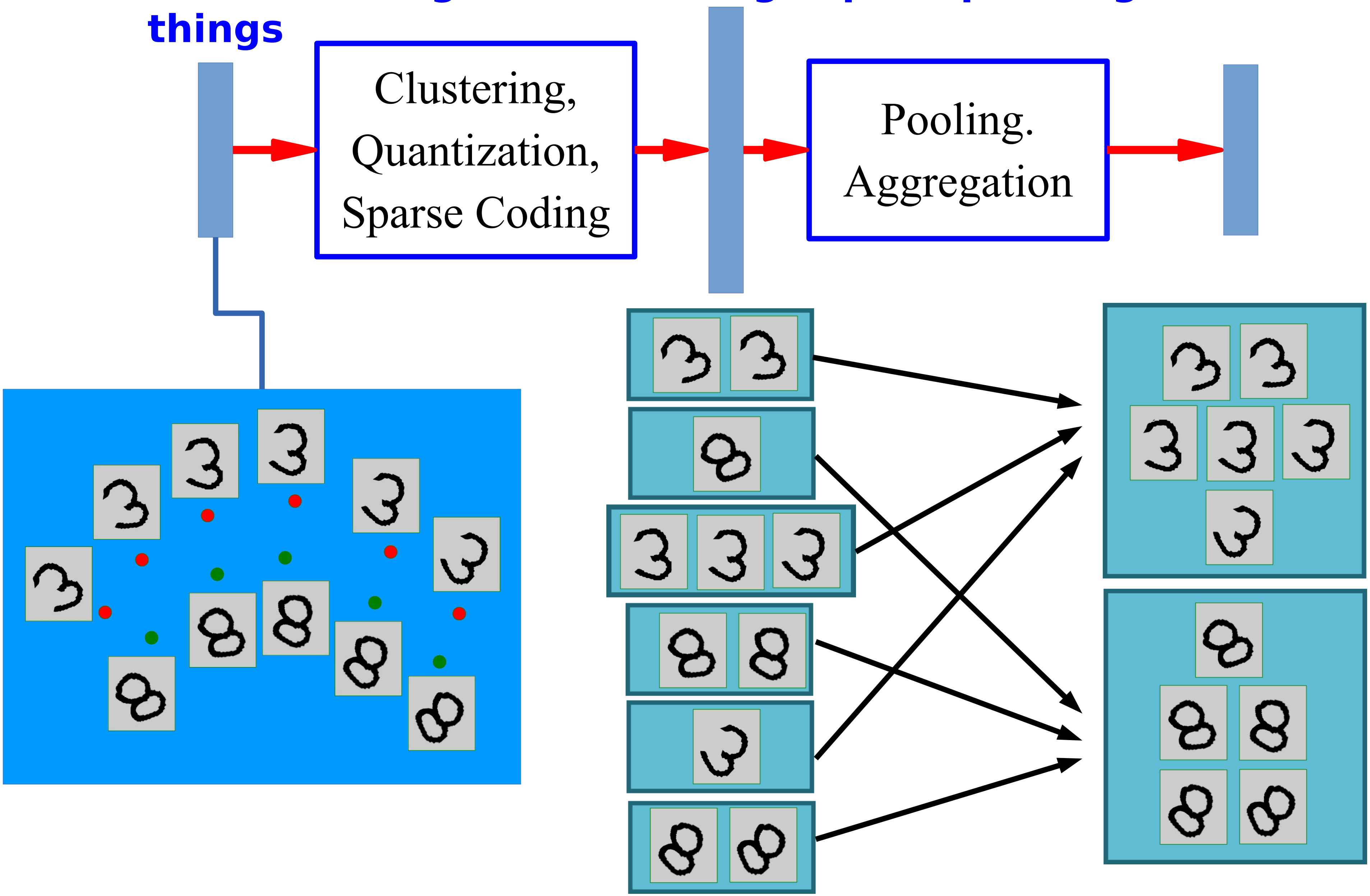
- **Entangled data manifolds**



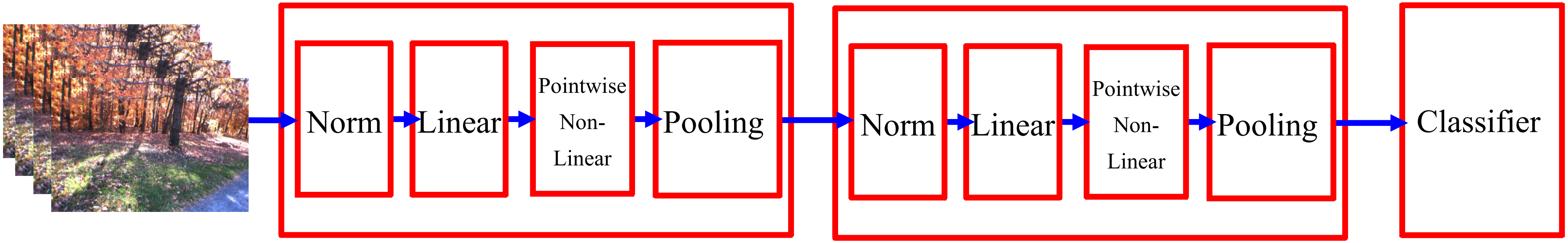
Sparse Non-Linear Expansion → Pooling

Y LeCun
MA Ranzato

- **Use clustering to break things apart, pool together similar things**



Overall Architecture: Normalization → Filter Bank → Non-Linearity → Pooling

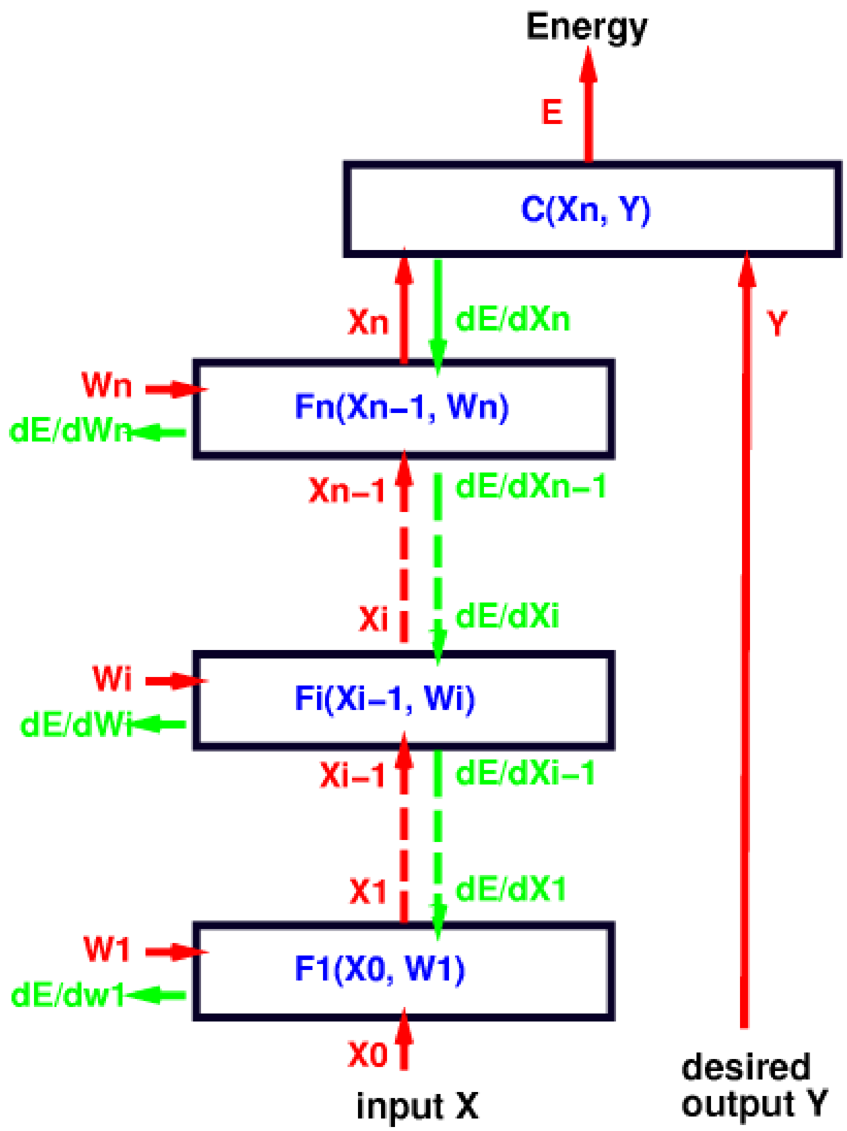


- **Stacking multiple stages of**
  - [Normalization → Filter Bank → Non-Linearity → Pooling].
- **Normalization: variations on whitening**
  - Subtractive: average removal, high pass filtering
  - Divisive: local contrast normalization, variance normalization
- **Linear: dimension expansion, projection on overcomplete basis**
- **(Pointwise) Non-Linear: Rectification, saturation....**
  - ReLU, Component-wise shrinkage, tanh, winner-takes-all
- **Pooling: aggregation over space or feature type**
  - $$X_i; \quad L_p: \sqrt[p]{X_i^p}; \quad PROB: \frac{1}{b} \log\left(\sum_i e^{bX_i}\right)$$

Deep Supervised Learning (modular approach)

Multimodule Systems: Cascade

Y LeCun
MA Ranzato



- **Complex learning machines can be built by assembling modules into networks**
- **Simple example: sequential/layered feed-forward architecture (cascade)**
- **Forward Propagation:**

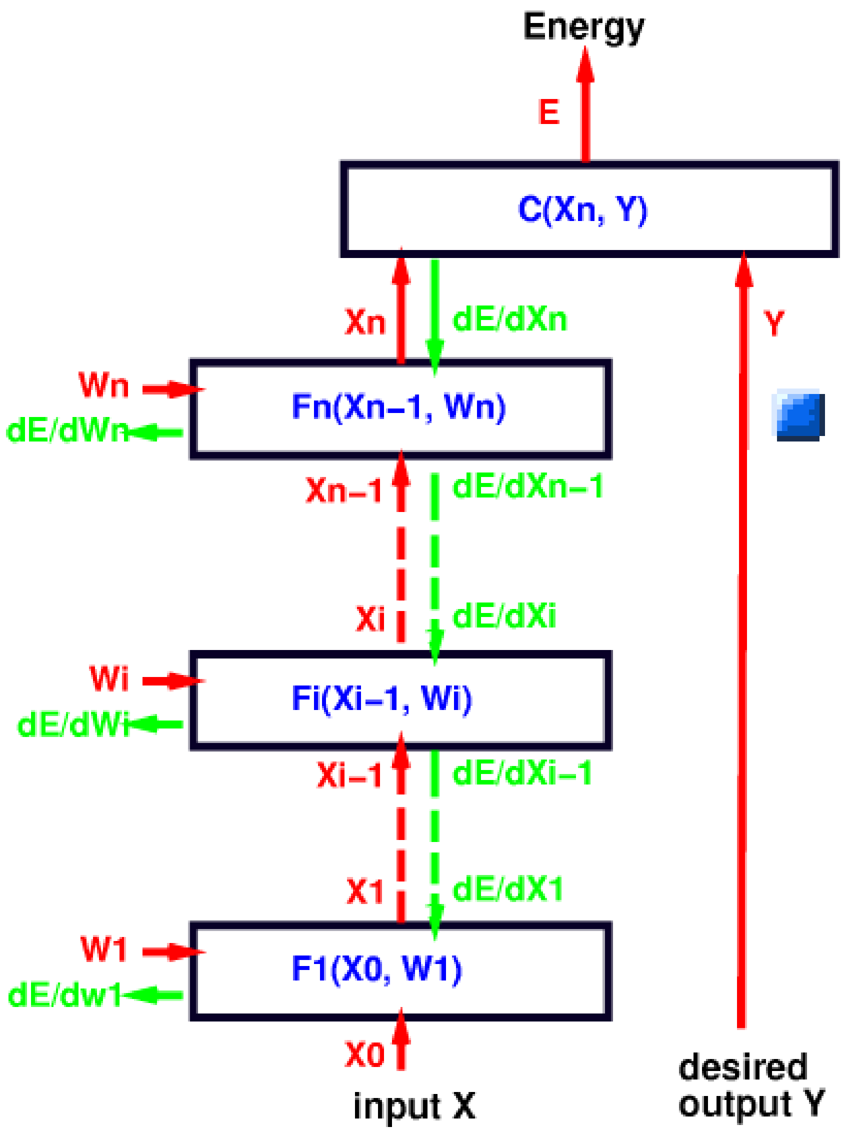
let $X = X_0$,

$$X_i = F_i(X_{i-1}, W_i) \quad \forall i \in [1, n]$$

$$E(Y, X, W) = C(X_n, Y)$$

Multimodule Systems: Implementation

Y LeCun
MA Ranzato



Each module is an object

- Contains trainable parameters
- Inputs are arguments
- Output is returned, but also stored internally
- Example: 2 modules m1, m2

PyTorch (functional paradigm)

- `m1 = nn.Linear(in_size, h_size)`
- `m2 = nn.Linear(h_size, out_size)`
- `# forward prop`
- `hid = torch.relu(m1(image.view(-1)))`
- `out = m2(hid)`

`image.view(-1)` flattens a tensor

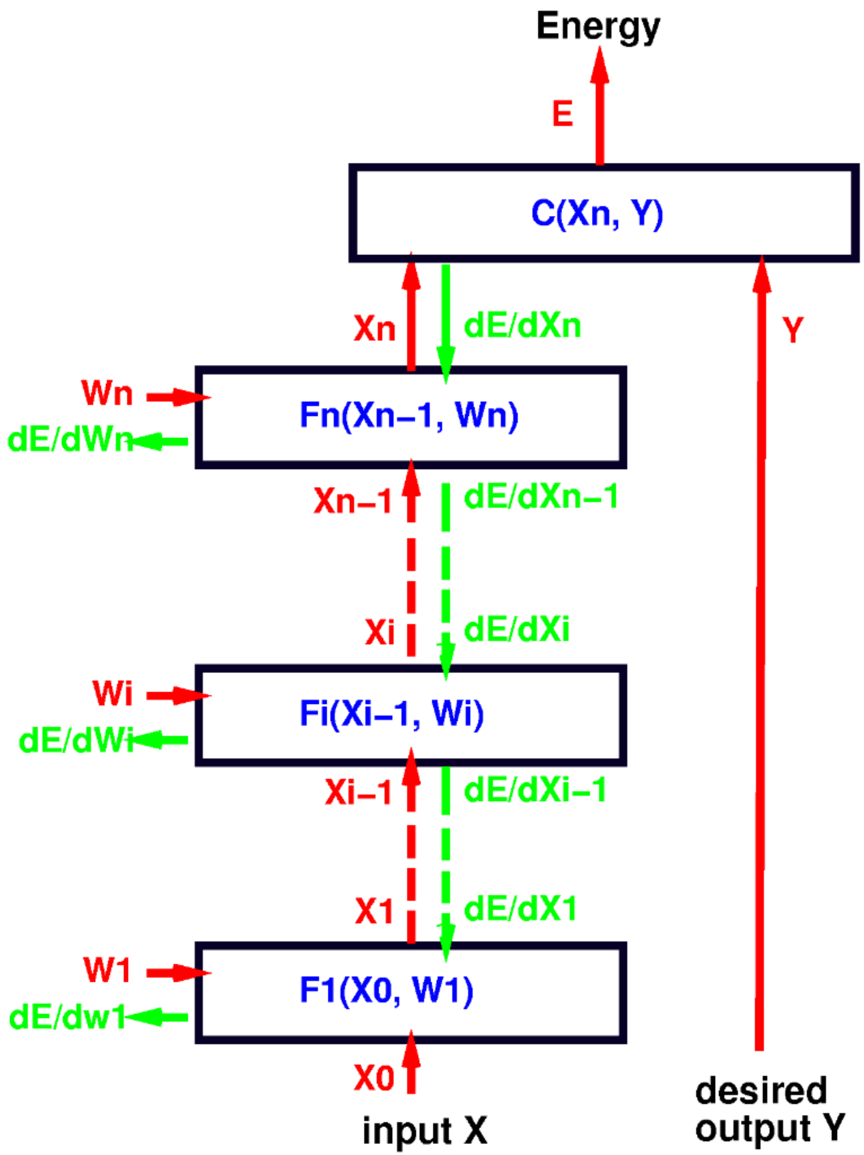
Multimodule Systems: Pytorch Implementation

Y LeCun
MA Ranzato

```
import torch
from torch import nn

image = torch.randn(3, 10, 20)
in_size = image.nelement()
h_size = 60
out_size = 6

#### Functional paradigm
m1 = nn.Linear(in_size, h_size)
m2 = nn.Linear(h_size, out_size)
# forward prop
hid = torch.relu(m1(image.view(-1)))
out = m2(hid)

#### Using containers
model = nn.Sequential(m1, nn.ReLU(), m2)
# forward prop
out = model(image.view(-1))

#### Using object oriented programming
class Net(nn.Module):
    def __init__(self, in_s, h_s, out_s):
        super().__init__()
        self.m1 = nn.Linear(in_s, h_s)
        self.m2 = nn.Linear(h_s, out_s)

    def forward(self, x):
        x = torch.relu(self.m1(x.view(-1)))
        x = self.m2(x)
        return x

model = Net(in_size, h_size, out_size)
out = model(image)
```

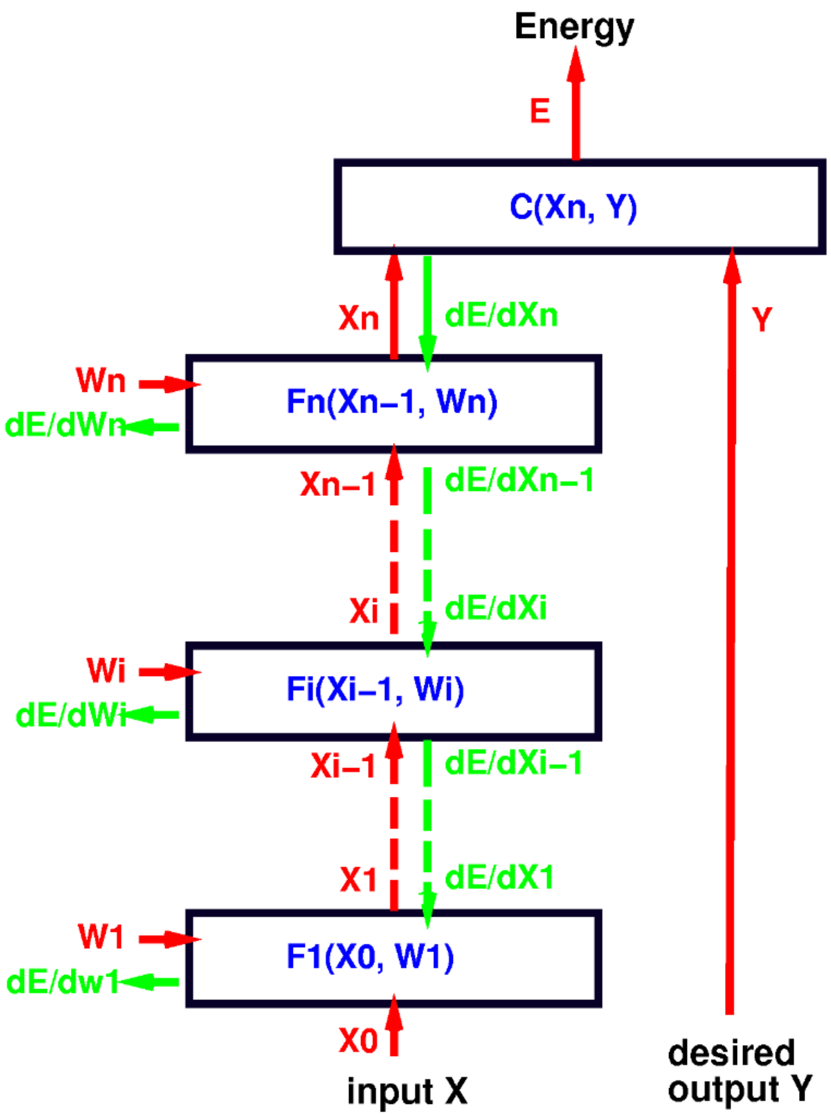
Computing the Gradient in Multi-Layer Systems

Y LeCun
MA Ranzato



- To train a multi-module system, we must compute the gradient of $E(W, Y, X)$ with respect to all the parameters in the system (all the W_k).
- Let's consider module i whose fprop method computes $X_k = F_k(X_{k-1}, W_k)$.
- Let's assume that we already know $\frac{\partial E}{\partial X_k}$, in other words, for each component of vector X_k we know how much E would wiggle if we wiggled that component of X_k.

Computing the Gradient in Multi-Layer Systems



- We can apply chain rule to compute $\frac{\partial E}{\partial W_k}$ (how much E would wiggle if we wiggled each component of W_k):

$$\frac{\partial E}{\partial W_k} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}$$

$$[1 \times N_w] = [1 \times N_x].[N_x \times N_w]$$

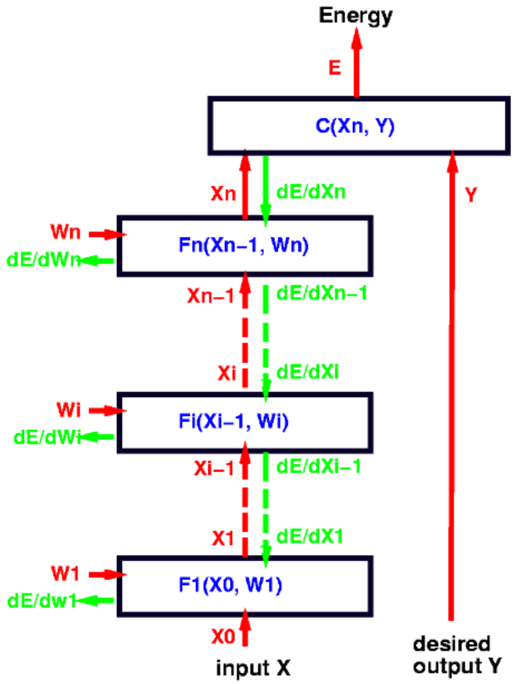
- $\frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}$ is the *Jacobian matrix* of F_k with respect to W_k.

$$\left[\frac{\partial F_k(X_{k-1}, W_k)}{\partial W_k}\right]_{pq} = \frac{\partial \left[F_k(X_{k-1}, W_k)\right]_p}{\partial [W_k]_q}$$

- Element (p, q) of the Jacobian indicates how much the p-th output wiggles when we wiggle the q-th weight.

Computing the Gradient in Multi-Layer Systems

Using the same trick, we can compute $\frac{\partial E}{\partial X_{k-1}}$. Let's assume again that we already know $\frac{\partial E}{\partial X_k}$, in other words, for each component of vector X_k we know how much E would wiggle if we wiggled that component of X_k.



- We can apply chain rule to compute $\frac{\partial E}{\partial X_{k-1}}$ (how much E would wiggle if we wiggled each component of X_{k-1}):

$$\frac{\partial E}{\partial X_{k-1}} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$$

- $\frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$ is the *Jacobian matrix* of F_k with respect to X_{k-1}.
- F_k has two Jacobian matrices, because it has to arguments.
- Element (p, q) of this Jacobian indicates how much the p-th output wiggles when we wiggle the q-th input.
- **The equation above is a recurrence equation!**

Jacobians and Dimensions

Y LeCun
MA Ranzato

- derivatives with respect to a column vector are line vectors (dimensions: $[1 \times N_{k-1}] = [1 \times N_k] * [N_k \times N_{k-1}]$)

$$\frac{\partial E}{\partial X_{k-1}} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}$$

- (dimensions: $[1 \times N_{wk}] = [1 \times N_k] * [N_k \times N_{wk}]$):

$$\frac{\partial E}{\partial W_k} = \frac{\partial E}{\partial X_k} \frac{\partial F_k(X_{k-1}, W_k)}{\partial W}$$

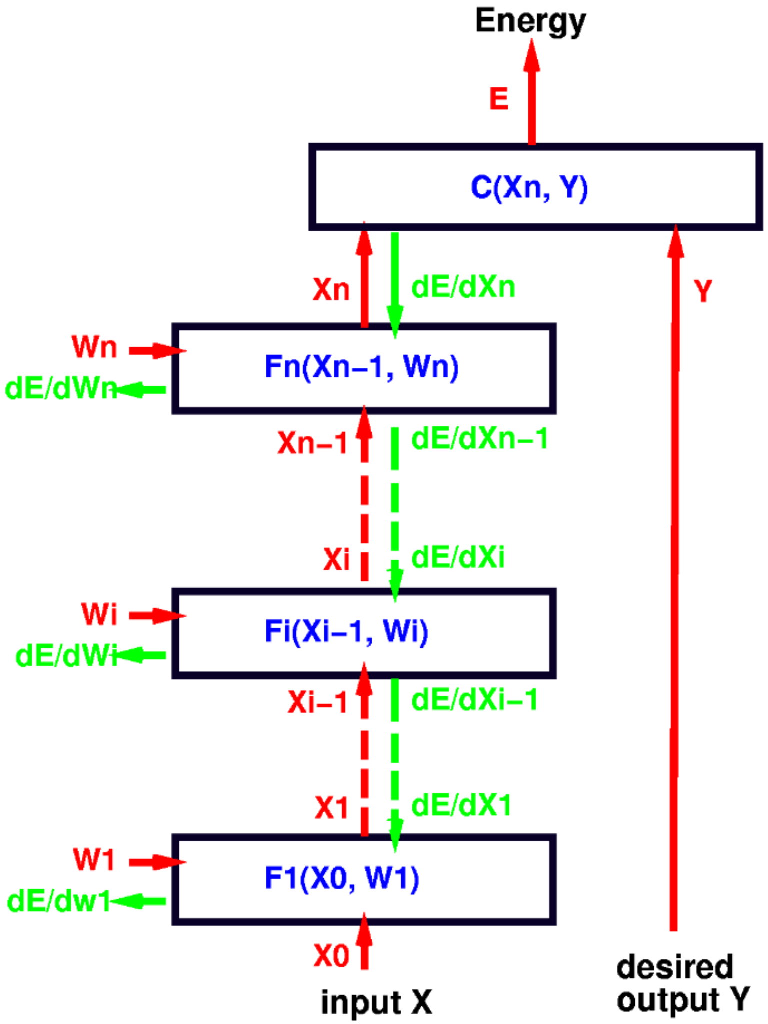
- we may prefer to write those equation with column vectors:

$$\frac{\partial E}{\partial X_{k-1}}' = \frac{\partial F_k(X_{k-1}, W_k)}{\partial X_{k-1}}' \frac{\partial E}{\partial X_k}'$$

$$\frac{\partial E}{\partial W_k}' = \frac{\partial F_k(X_{k-1}, W_k)}{\partial W}' \frac{\partial E}{\partial X_k}'$$

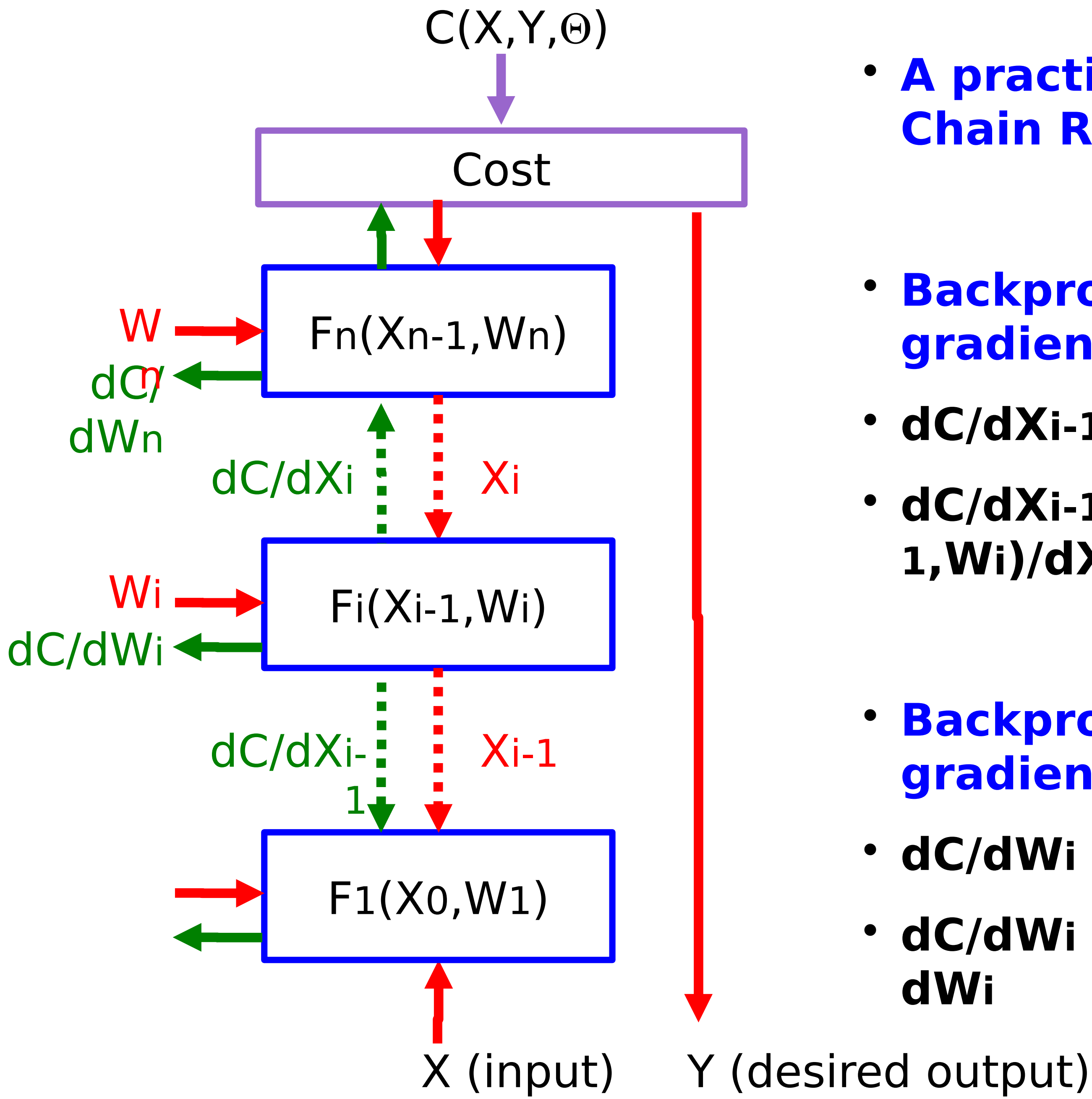
Back Propgation

To compute all the derivatives, we use a backward sweep called the **back-propagation algorithm** that uses the recurrence equation for $\frac{\partial E}{\partial X_k}$



- $\frac{\partial E}{\partial X_n} = \frac{\partial C(X_n,Y)}{\partial X_n}$
- $\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1},W_n)}{\partial X_{n-1}}$
- $\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1},W_n)}{\partial W_n}$
- $\frac{\partial E}{\partial X_{n-2}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2},W_{n-1})}{\partial X_{n-2}}$
- $\frac{\partial E}{\partial W_{n-1}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2},W_{n-1})}{\partial W_{n-1}}$
- ....etc, until we reach the first module.
- we now have all the $\frac{\partial E}{\partial W_k}$ for $k \in [1, n]$.

Computing Gradients by Back-Propagation

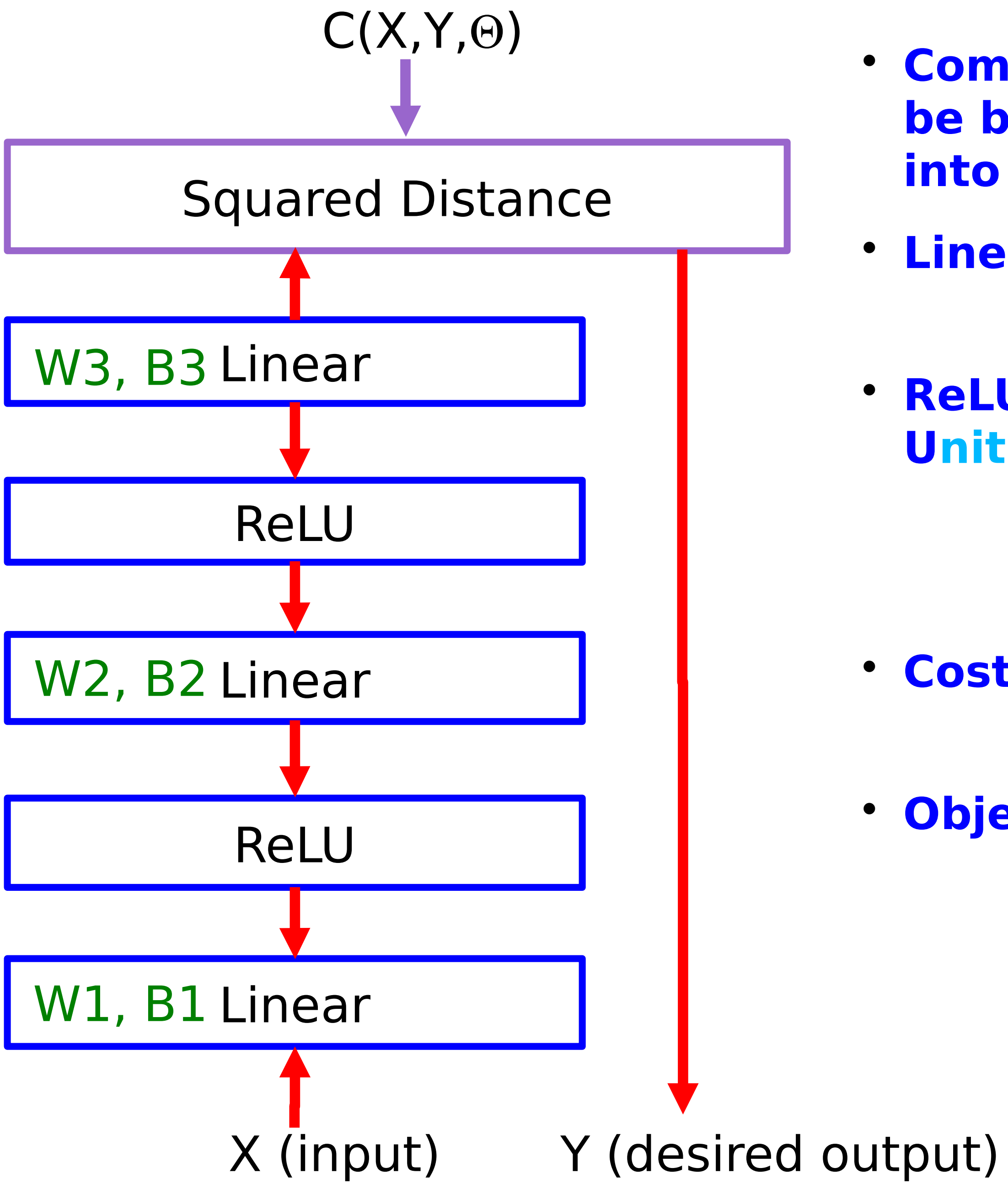


- **A practical Application of Chain Rule**

- **Backprop for the state gradients:**
- **$dC/dX_{i-1} = dC/dX_i \, . \, dX_i/dX_{i-1}$**
- **$dC/dX_{i-1} = dC/dX_i \, . \, dF_i(X_{i-1},W_i)/dX_{i-1}$**

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

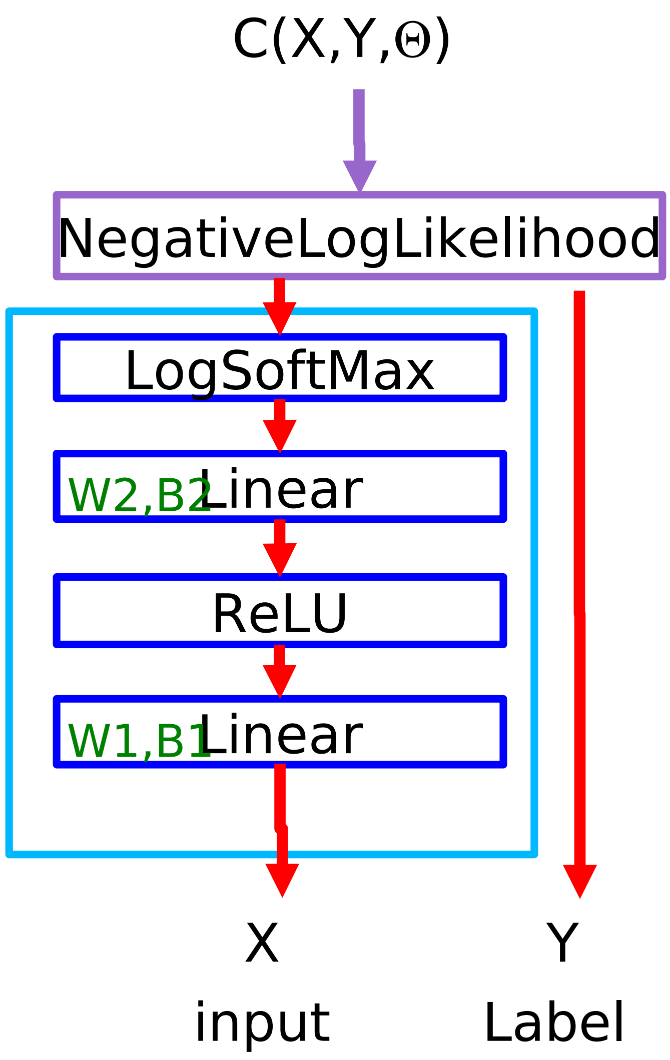
Typical Multilayer Neural Net Architecture



- **Complex learning machines can be built by assembling modules into networks**
- **Linear Module**
  - Out = W.In+B
- **ReLU Module (Rectified Linear Unit)**
  - $Out_i = 0$ if $In_i<0$
  - $Out_i = In_i$ otherwise
- **Cost Module: Squared Distance**
  - $C = ||In1 - In2||^2$
- **Objective Function**
  - $L(\Theta)=1/p\ \Sigma_k\ C(X^k,Y^k,\Theta)$
  - $\Theta = (W1,B1,W2,B2,W3,B3)$

Building a Network by Assembling Modules

- All deep learning frameworks use modules (inspired by SN/Lush, 1991)
  - PyTorch, TensorFlow….



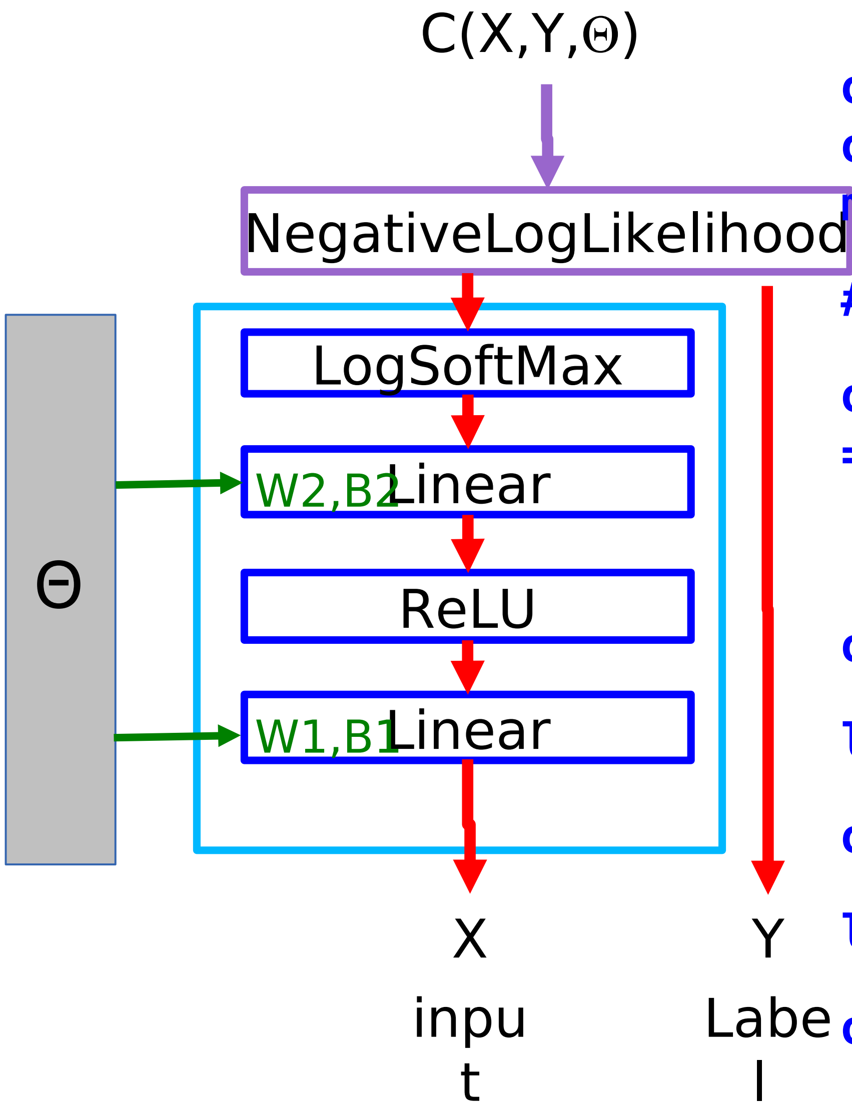
PyTorch

```
class Net(nn.Module):
    def __init__(self,insize,hsize,outsize):
        super().__init__()
        self.m1 = nn.Linear(insize,hsize)
        self.m2 = nn.Linear(hsize,outsize)

    def forward(self, x):
        x = F.relu(self.m1(x))
        x = self.m2(x)
        return F.logsoftmax(x)

model = Net(784,500,10)
model(image)
```

Running Backprop

Y LeCun

- PyTorch example



PyTorch

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

# or

optimizer = optim.Adam([var1, var2], lr = 0.0001)


output = model(image)

loss = F.nll_loss(output,target)

optimizer.zero_grad()

loss.backward()

optimizer.step()
```

Module Classes

Linear
- $Y = W.X$; $dC/dX = W^T . dC/dY$; $dC/dW = dC/dY . X^T$

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
- $Yi = Xi - \log\left[\sum_j \exp(Xj)\right]$; .....