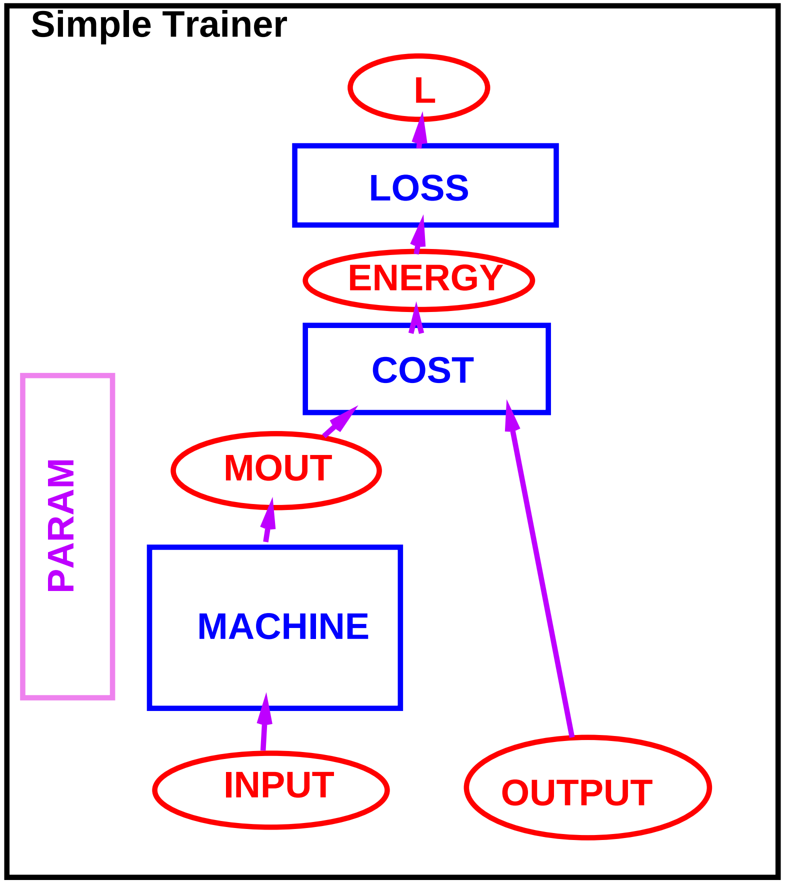
MACHINE LEARNING AND PATTERN RECOGNITION

Spring 2004, Lecture 4b

Modules and Architectures

Yann LeCun
The Courant Institute,
New York University
http://yann.lecun.com

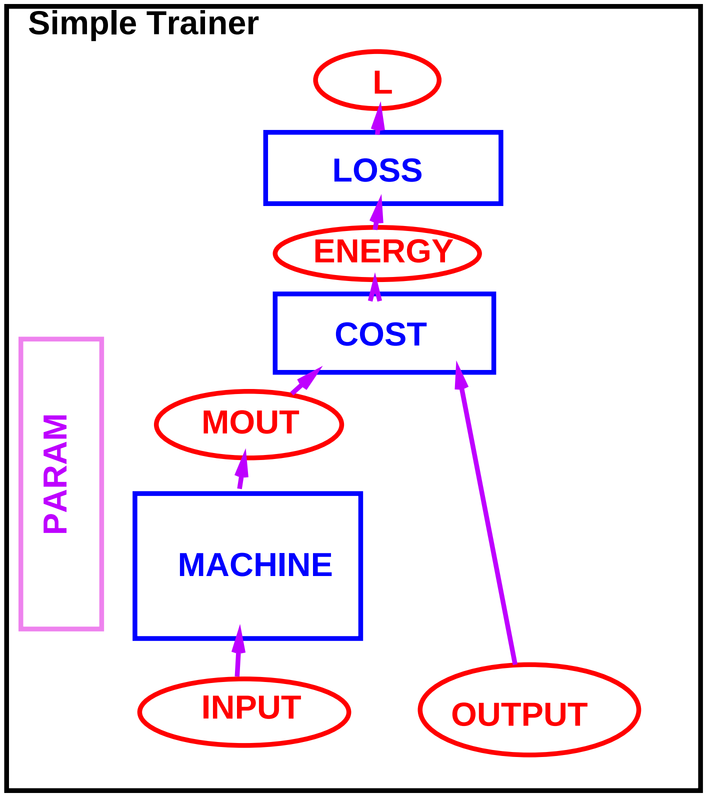
A Trainer class



The trainer object is designed to train a particular machine with a given energy function and loss. The example below uses the simple energy loss.

```
(defclass simple-trainer object
  input ; the input state
  output ; the output/label state
  machin ; the machine
  mout ; the output of the machine
  cost ; the cost module
  energy ; the energy (output of the cost) and
  param ; the trainable parameter vector
  )
```

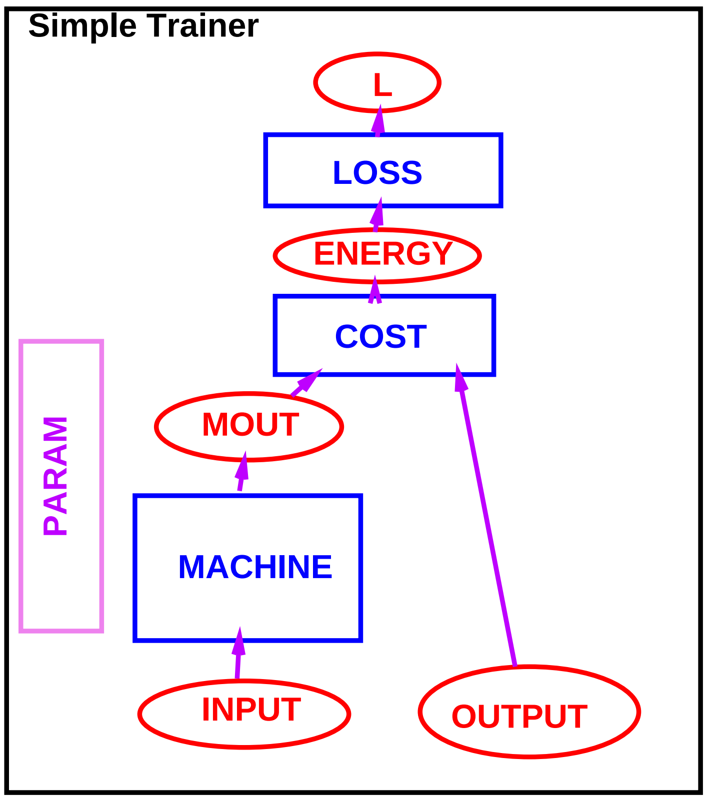
A Trainer class: running the machine



Takes an input and a vector of possible labels (each of which is a vector, hence <label-set> is a matrix) and returns the index of the label that minimizes the energy. Fills up the vector <energies> with the energy produced by each possible label.

```
(defmethod simple-trainer run
      (sample label-set energies)
  (==> input resize (idx-dim sample 0))
  (idx-copy sample :input:x)
  (==> machine fprop input mout)
  (idx-bloop ((label label-set) (e energies))
    (==> output resize (idx-dim label 0))
    (idx-copy label :output:x)
    (==> cost fprop mout output energy)
    (e (:energy:x)))
  ;; find index of lowest energy
  (idx-d1indexmin energies))
```

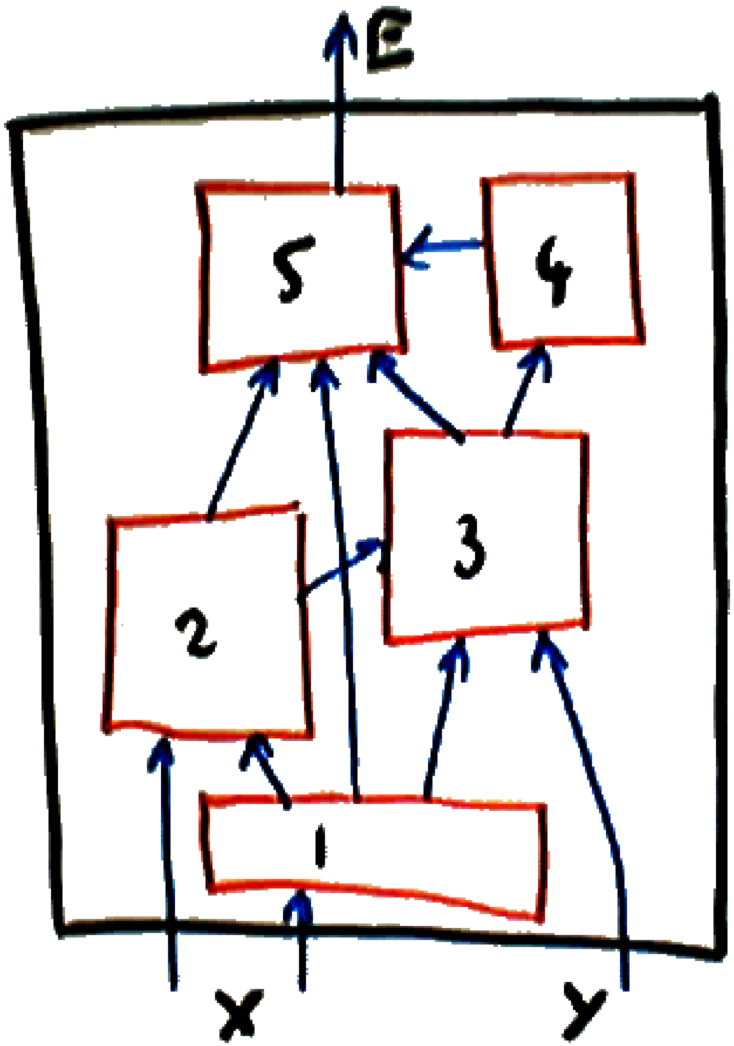
A Trainer class: training the machine



Performs a learning update on one sample. <sample> is the input sample, <label> is the desired category (an integer), <label-set> is a matrix where the i-th row is the desired output for the i-th category, and <update-args> is a list of arguments for the parameter update method (e.g. learning rate and weight decay).

```
(defmethod simple-trainer learn-sample
      (sample label label-set update-args)
  (==> input resize (idx-dim sample 0))
  (idx-copy sample :input:x)
  (==> machine fprop input mout)
  (==> output resize (idx-dim label-set 1))
  (idx-copy (select label-set 0 (label 0)) :output
  (==> cost fprop mout output energy)
  (==> cost bprop mout output energy)
  (==> machine bprop input mout)
  (==> param update update-args)
  (:energy:x))
```

Other Topologies



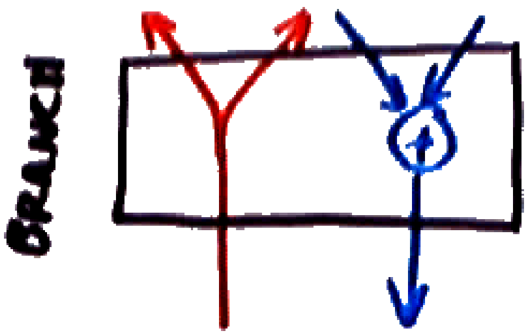
- The back-propagation procedure is not limited to feed-forward cascades.
- It can be applied to networks of module with *any* topology, as long as the connection graph is acyclic.
- If the graph is acyclic (no loops) then, we can easily find a suitable order in which to call the fprop method of each module.
- The bprop methods are called in the reverse order.
- if the graph has cycles (loops) we have a so-called *recurrent network*. This will be studied in a subsequent lecture.

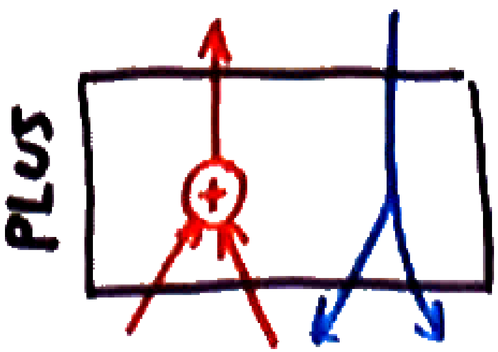
More Modules

A rich repertoire of learning machines can be constructed with just a few module types in addition to the linear, sigmoid, and euclidean modules we have already seen.
We will review a few important modules:

- The branch/plus module
- The switch module
- The Softmax module
- The logsum module

The Branch/Plus Module





- The PLUS module: a module with K inputs $X_1, \ldots, X_K$ (of any type) that computes the sum of its inputs:

$$X_{\text{out}} = \sum_k X_k$$

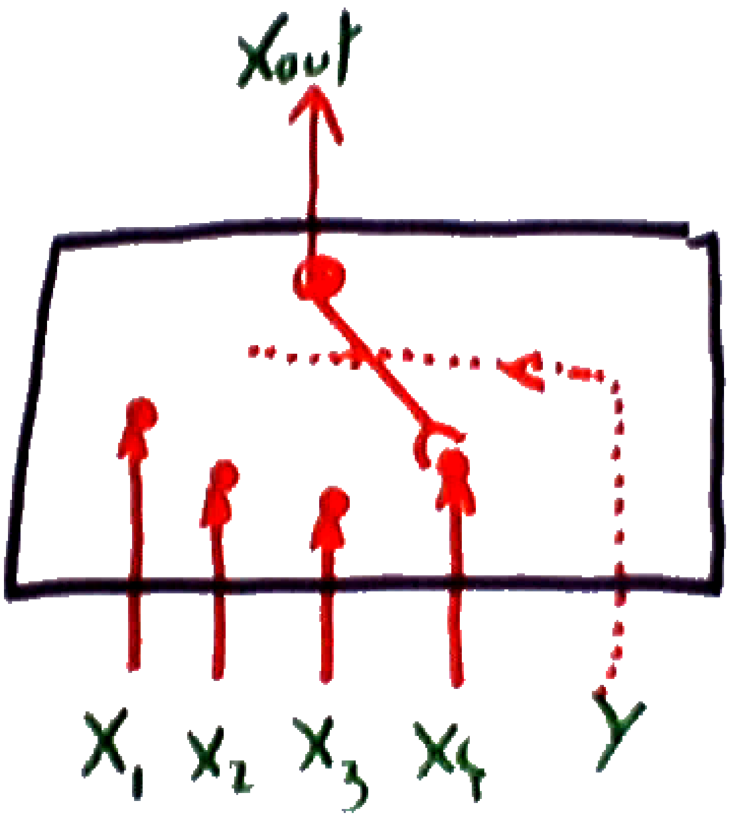
back-prop: $\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial X_{\text{out}}} \quad \forall k$

- The BRANCH module: a module with one input and K outputs $X_1, \ldots, X_K$ (of any type) that simply copies its input on its outputs:

$$X_k = X_{\text{in}} \quad \forall k \in [1..K]$$

back-prop: $\frac{\partial E}{\partial \text{in}} = \sum_k \frac{\partial E}{\partial X_k}$

The Switch Module



- A module with K inputs $X_1, \ldots, X_K$ (of any type) and one additional discrete-valued input Y.
- The value of the discrete input determines which of the N inputs is copied to the output.
- $$X_{\text{out}} = \sum_k \delta(Y-k) X_k$$

$$\frac{\partial E}{\partial X_k} = \delta(Y-k) \frac{\partial E}{\partial X_{\text{out}}}$$

the gradient with respect to the output is copied to the gradient with respect to the switched-in input. The gradients of all other inputs are zero.

The Logsum Module

fprop:

$$X_{\text{out}} = -\frac{1}{\beta} \log \sum_k \exp(-\beta X_k)$$

bprop:

$$\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial X_{\text{out}}} \frac{\exp(-\beta X_k)}{\sum_j \exp(-\beta X_j)}$$

or

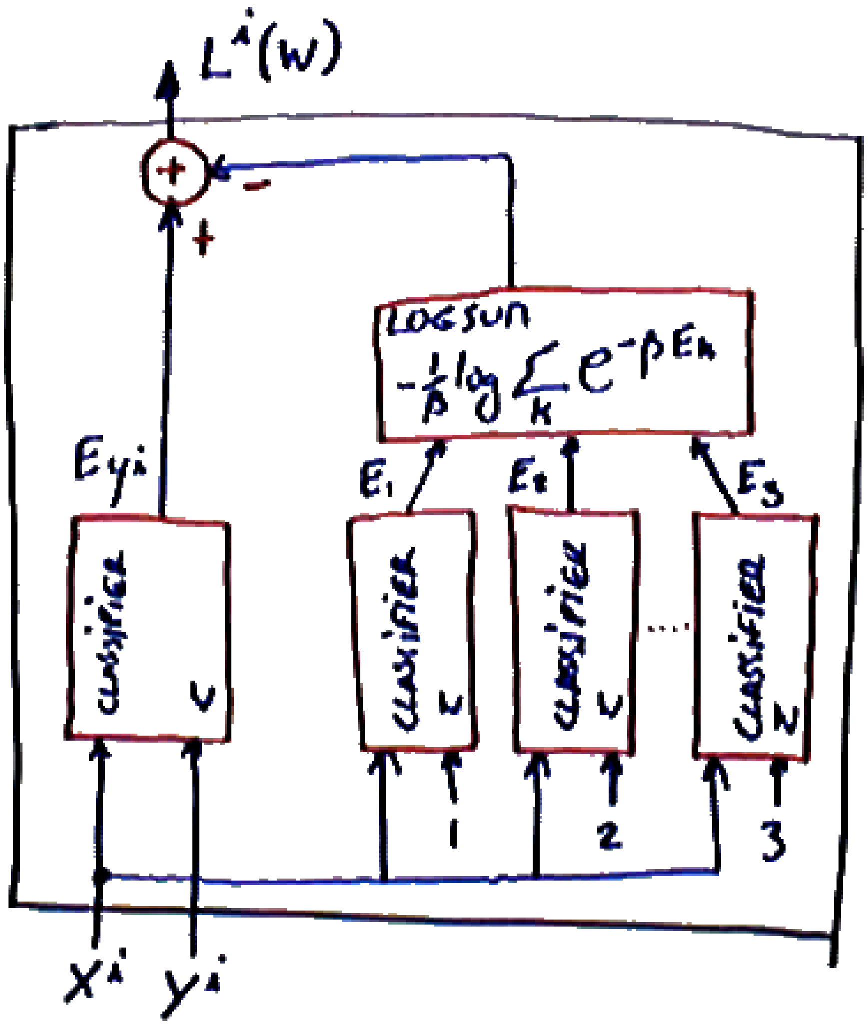
$$\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial X_{\text{out}}} P_k$$

with

$$P_k = \frac{\exp(-\beta X_k)}{\sum_j \exp(-\beta X_j)}$$

Log-Likelihood Loss function and Logsum Modules

MAP/MLE Loss $L_{\text{ll}}(W, Y^i, X^i) = E(W, Y^i, X^i) + \frac{1}{\beta} \log \sum_k \exp(-\beta E(W, k, X^i))$



- A classifier trained with the Log-Likelihood loss can be transformed into an equivalent machine trained with the energy loss.
- The transformed machine contains multiple “replicas” of the classifier, one replica for the desired output, and K replicas for each possible value of Y.

Softmax Module

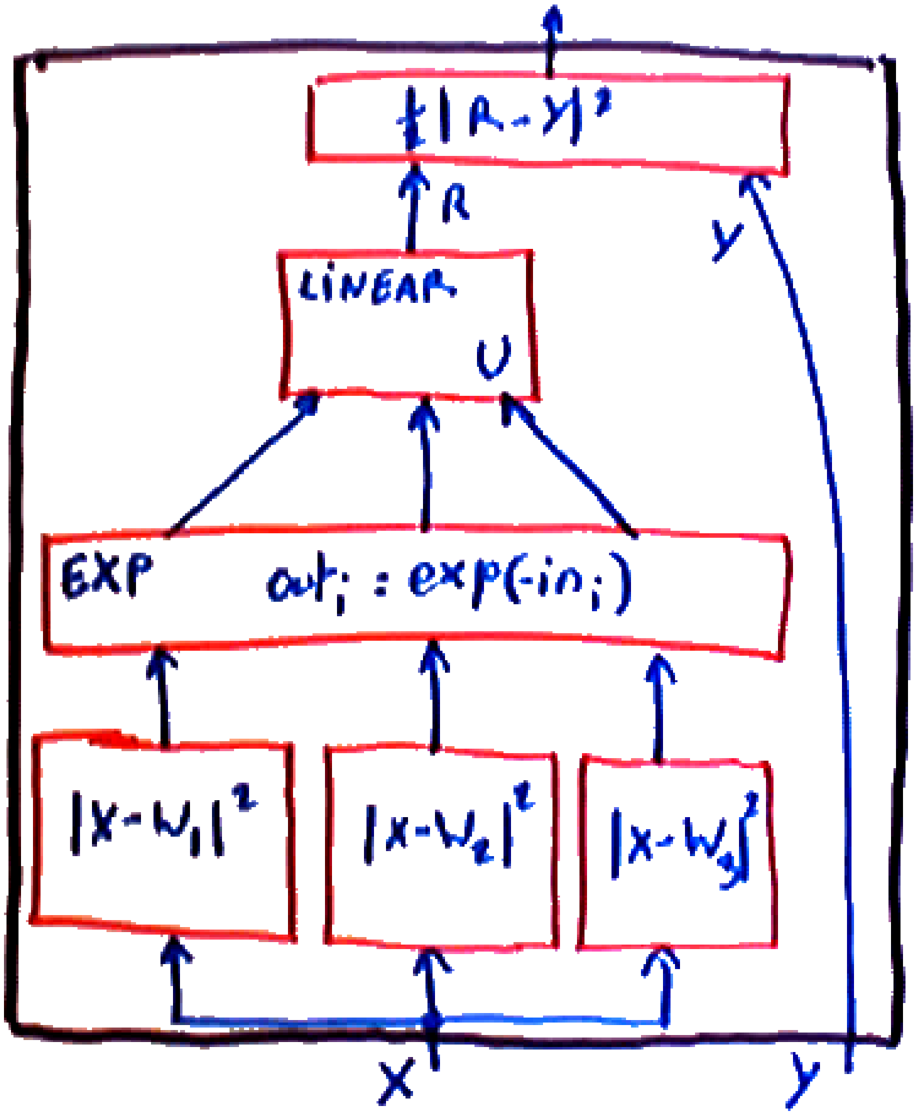
A single vector as input, and a “normalized” vector as output:

$$(X_{\text{out}})_i = \frac{\exp(-\beta x_i)}{\sum_k \exp(-\beta x_k)}$$

Exercise: find the bprop

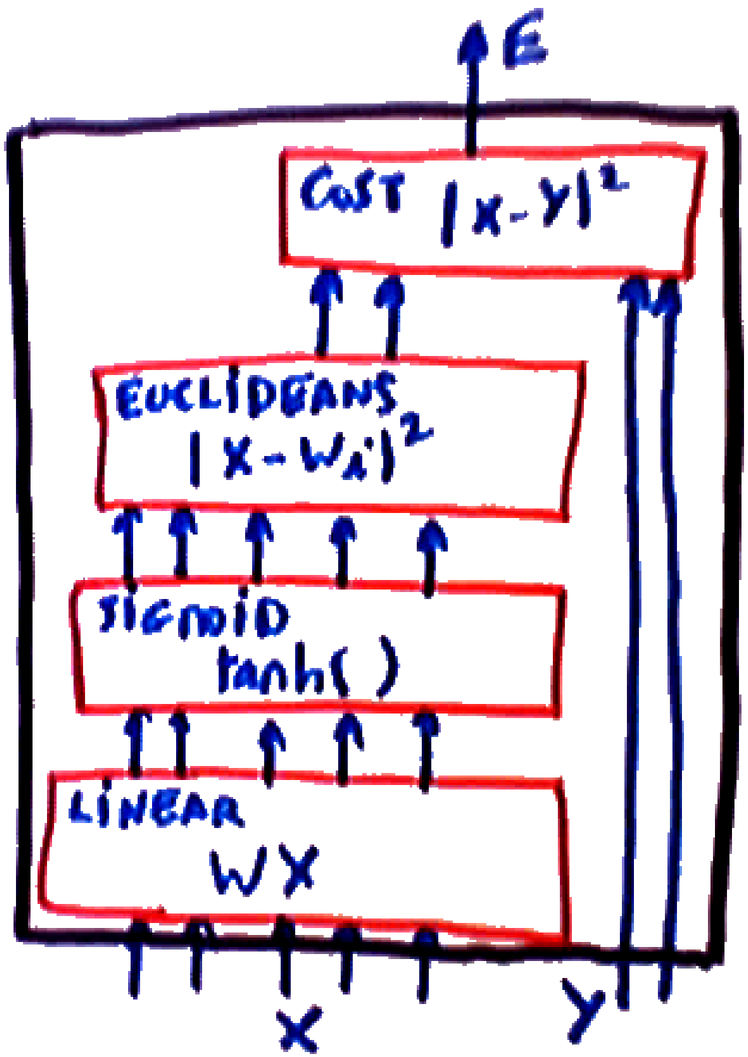
$$\frac{\partial (X_{\text{out}})_i}{\partial x_j} = ???$$

Radial Basis Function Network (RBF Net)



- Linearly combined Gaussian bumps.
- $F(X, W, U) = \sum_i u_i \exp(-k_i (X - W_i)^2)$
- The centers of the bumps can be initialized with the K-means algorithm (see below), and subsequently adjusted with gradient descent.
- This is a good architecture for regression and function approximation.

NN-RBF Hybrids

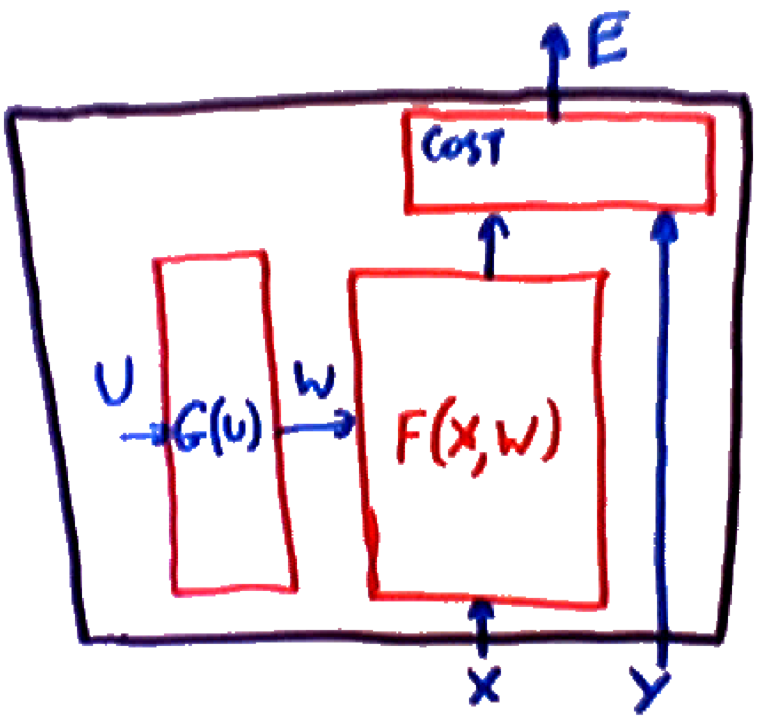


- sigmoid units are generally more appropriate for low-level feature extraction.
- Euclidean/RBF units are generally more appropriate for final classifications, particularly if there are many classes.
- Hybrid architecture for multiclass classification: sigmoids below, RBFs on top + softmax + log loss.

Parameter-Space Transforms

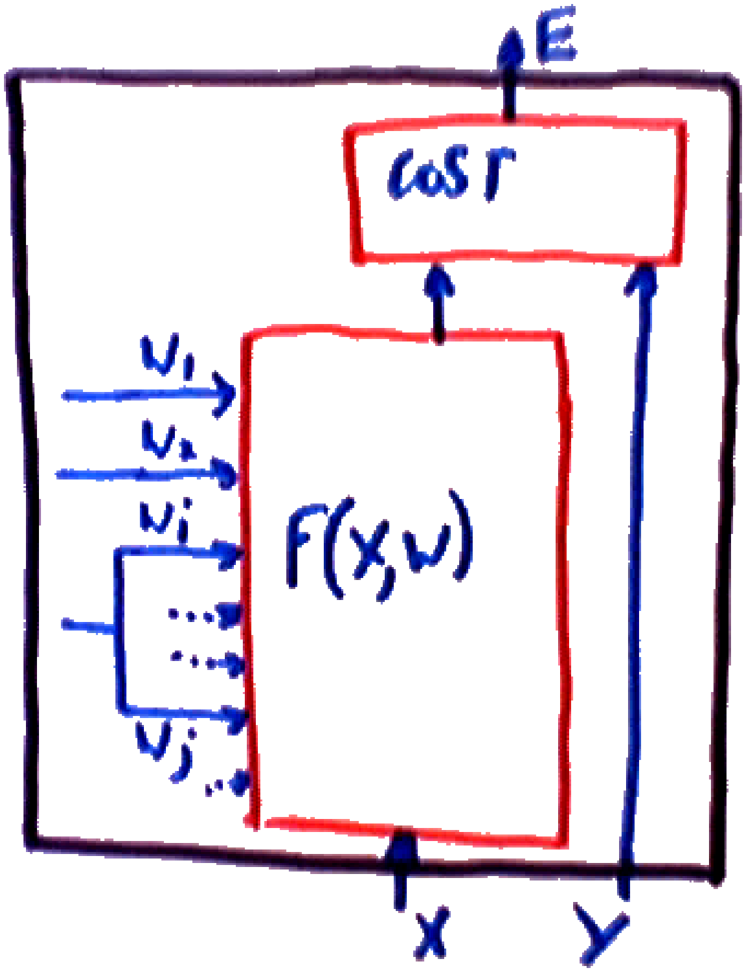
Reparameterizing the function by transforming the space

$$E(Y, X, W) \rightarrow E(Y, X, G(U))$$



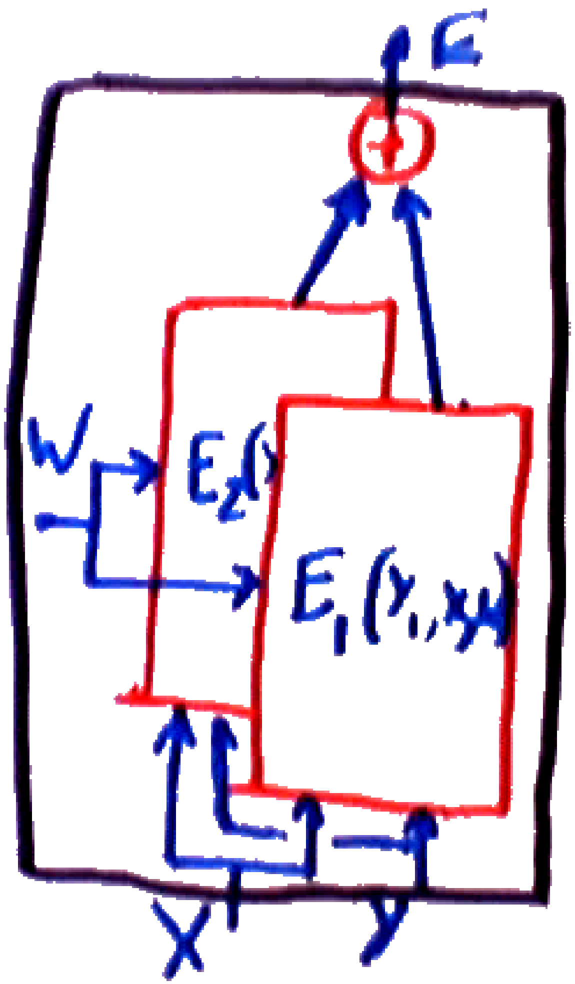
- gradient descent in U space: $U \leftarrow U - \eta \frac{\partial G}{\partial U}' \frac{\partial E(Y,X,W)}{\partial W}'$
- equivalent to the following algorithm in W space: $W \leftarrow W - \eta \frac{\partial G}{\partial U} \frac{\partial G}{\partial U}' \frac{\partial E(Y,X,W)}{\partial W}'$
- dimensions: $[N_w \times N_u][N_u \times N_w][N_w]$

Parameter-Space Transforms: Weight Sharing



- A single parameter is replicated multiple times in a machine
- $E(Y, X, w_1, \ldots, w_i, \ldots, w_j, \ldots) \rightarrow E(Y, X, w_1, \ldots, u_k, \ldots, u_k, \ldots)$
- gradient: $\frac{\partial E()}{\partial u_k} = \frac{\partial E()}{\partial w_i} + \frac{\partial E()}{\partial w_j}$
- w_i and w_j are tied, or equivalently, u_k is shared between two locations.

Parameter Sharing between Replicas



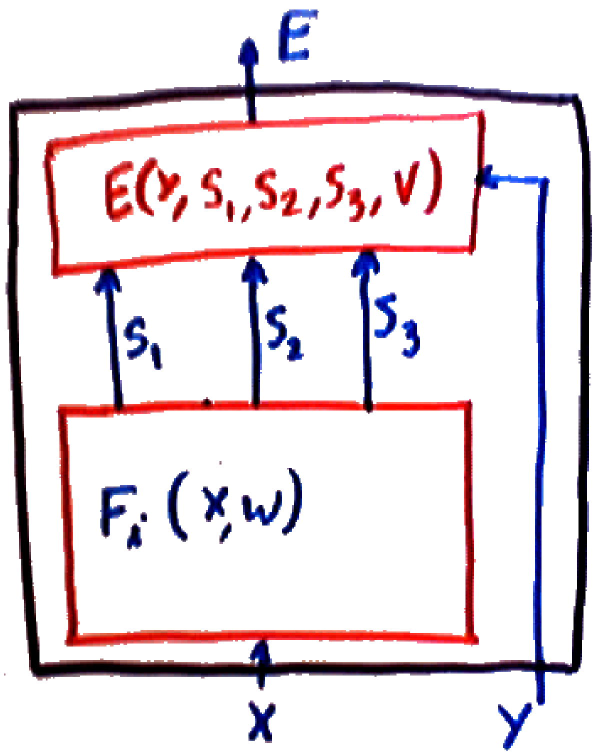
- We have seen this before: a parameter controls several replicas of a machine.
-

$$E(Y_1, Y_2, X, W) = E_1(Y_1, X, W) + E_1(Y_2, X, W)$$

- gradient:
$\frac{\partial E(Y_1,Y_2,X,W)}{\partial W} = \frac{\partial E_1(Y_1,X,W)}{\partial W} + \frac{\partial E_1(Y_2,X,W)}{\partial W}$
- W is shared between two (or more) instances of the machine: just sum up the gradient contributions from each instance.

Path Summation (Path Integral)

One variable influences the output through several others



- $E(Y, X, W) =$ $E(Y, F_1(X, W), F_2(X, W), F_3(X, W), V)$
- gradient: $\frac{\partial E(Y,X,W)}{\partial X} = \sum_i \frac{\partial E_i(Y,S_i,V)}{\partial S_i} \frac{\partial F_i(X,W)}{\partial X}$
- gradient: $\frac{\partial E(Y,X,W)}{\partial W} = \sum_i \frac{\partial E_i(Y,S_i,V)}{\partial S_i} \frac{\partial F_i(X,W)}{\partial W}$
- there is no need to implement these rules explicitely. They come out naturally of the object-oriented implementation.