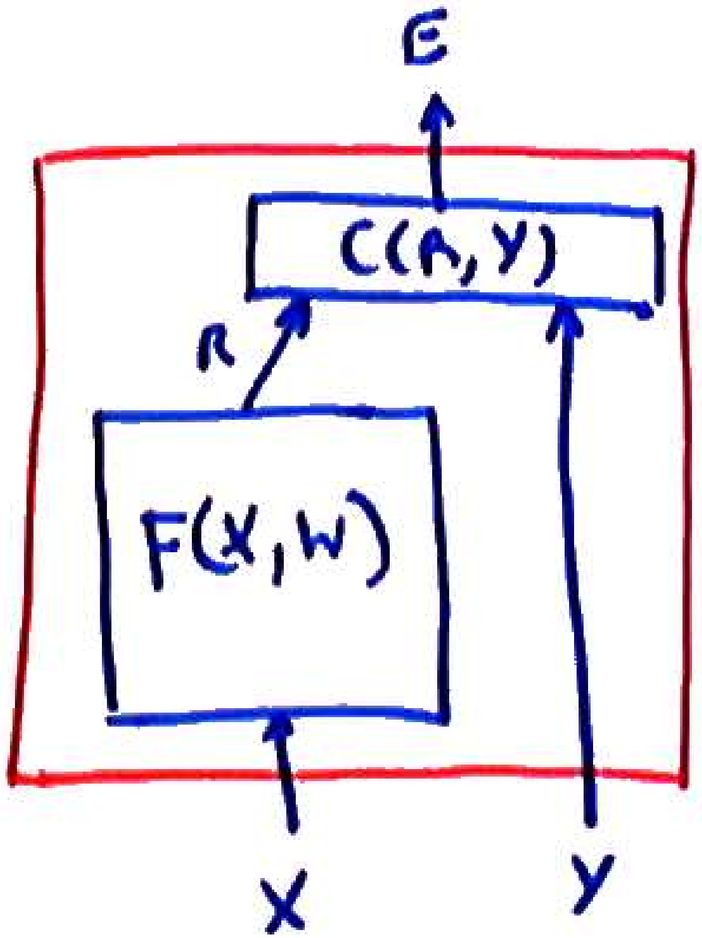
MACHINE LEARNING AND PATTERN RECOGNITION

Spring 2004, Lecture 5: Gradient-Based Learning II: Multi-Layer Nets and Back-propagation.

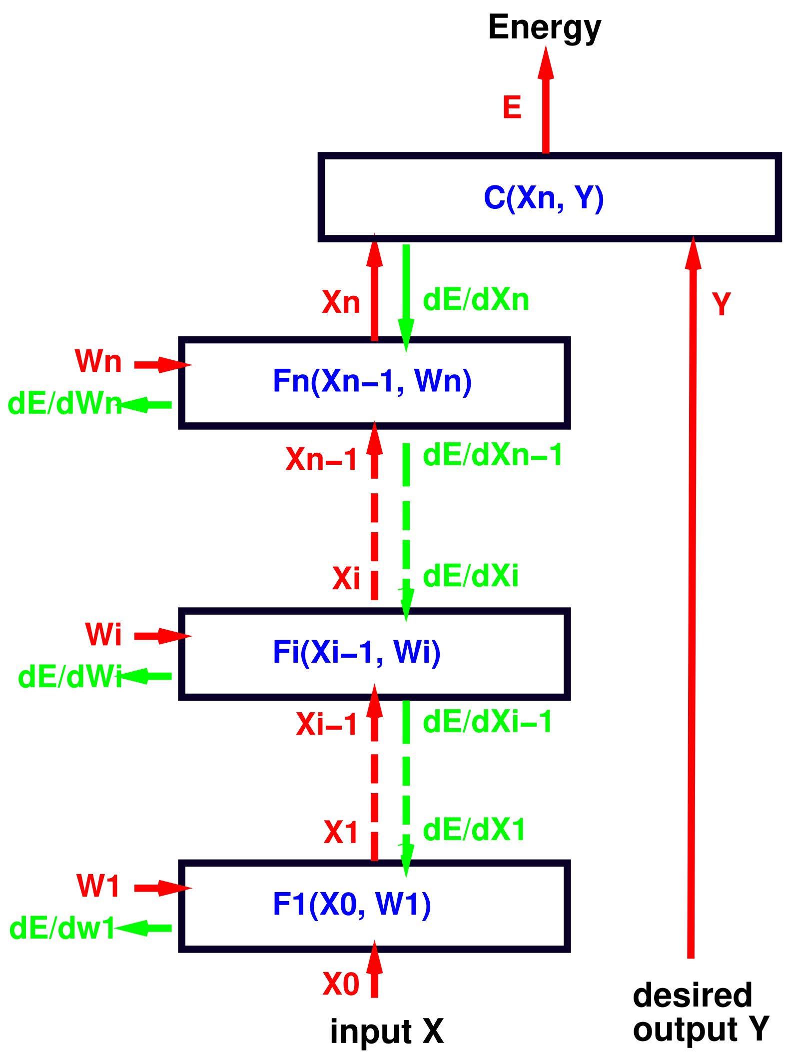
Yann LeCun
The Courant Institute,
New York University
http://yann.lecun.com

Non-Linear Learning



- So far, we have seen how to rain linear machines, and we have hinted at the fact that we could also train non-linear machines.
- In non-linear machine, the discriminant function $F(X, W)$ is allowed to be *non linear* with respect to W and *non linear* with respect to X.
- This allows us play with a much larger set of parameterized families of functions with a rich repertoire of class boundaries.
- well-designed non-linear classifiers can learn complex boundaries and take care of complicated intra-class variabilitites.

Multi-Module Systems

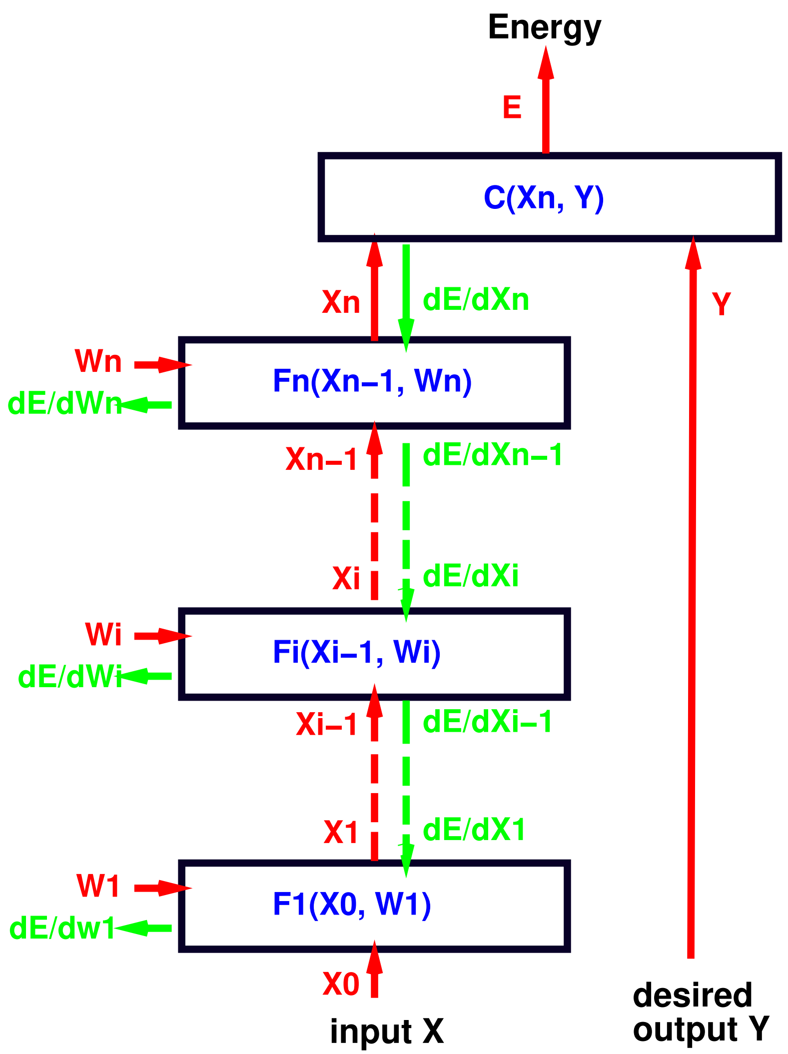


- Complex learning machines can be built by assembling Modules into networks.
- a simple example: layered, feed-forward architecture (cascade).
- computing the output from the input: forward propagation
- let $X = X_0$,

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

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

Object-Oriented Implementation



- Each module is an object (instance of a class).
- Each class has an "fprop" (forward propagation) method that takes the input and output states as arguments and computes the output state from the input state.
- Lush:
  ```
  (==> module fprop input output)
  ```
- C++:
  ```
  module.fprop(input,output);
  ```

Gradient-Based Learning in Multi-Module Systems

- Learning comes down to finding the W that minimizes the following objective function averaged over the training set $\{(X^1, Y^1), (X^2, Y^2), \ldots, (X^P, Y^P)\}$.

$$L(W) = \frac{1}{P}\sum_{i=1}^{P}[E(Y^i, X^i, W) + \frac{1}{\beta}\log\int \exp(-\beta E(Y, X^i, W))dY] + \frac{\lambda}{P}H(W)$$

$$L(W) = \frac{1}{P}\sum_{i=1}^{P}[E(Y^i, X^i, W) - E^*(X^i, W)] + \frac{\lambda}{P}H(W)$$

- λ is an appropriately picked coefficient that determines the importance of the regularization term.
- We will allow ourselves to choose $E^*(X^i, W)$ as we please (with no theoretical guarantee that our choice will be valid!).

Gradient-Based Learning in Multi-Module Systems

- Batch gradient descent (compute the full gradient before an update):

$$W \leftarrow W - \frac{\eta}{P}\left[\frac{\partial \sum_i E(Y^i, X^i, W) - E^*(X^i, W)}{\partial W} + \frac{\partial H(W)}{\partial W}\right]$$

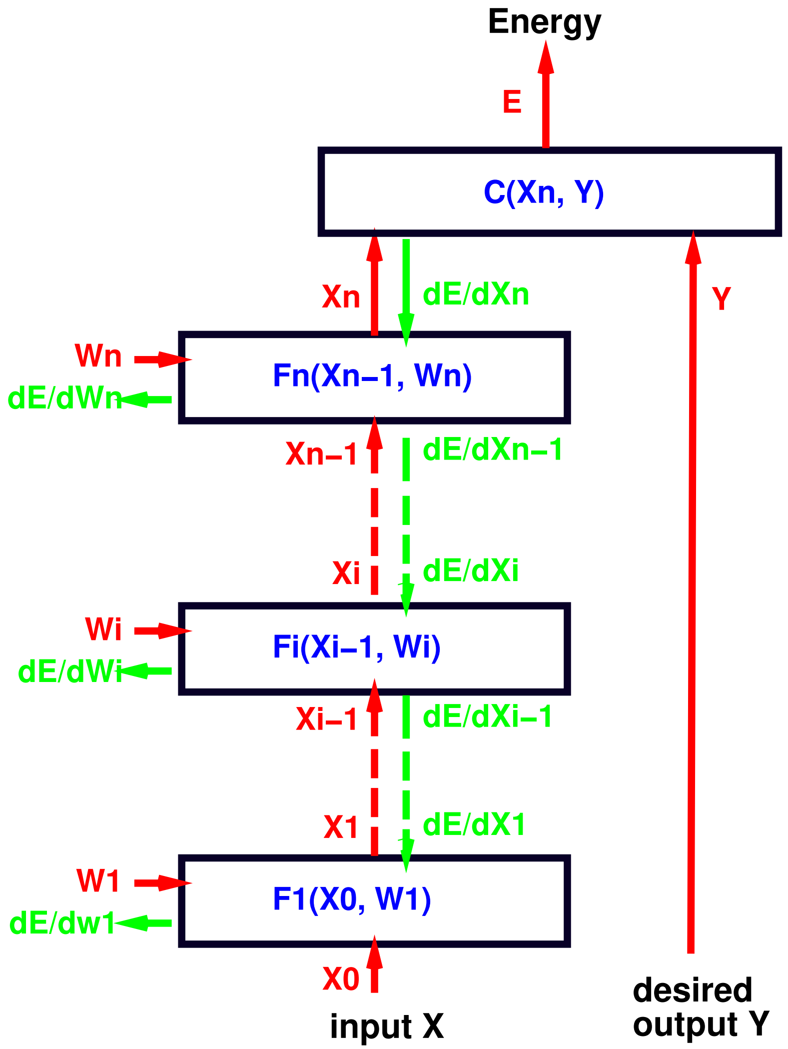
$$W \leftarrow W - \frac{\eta}{P}\left[\sum_i \left(\frac{\partial E(Y^i, X^i, W)}{\partial W} - \frac{\partial E^*(X^i, W)}{\partial W}\right) + \frac{\partial H(W)}{\partial W}\right]$$

- On-Line gradient descent (compute the gradient for one sample, and update)

$$W \leftarrow W - \eta\left[\frac{\partial E(Y^i, X^i, W)}{\partial W} - \frac{\partial E^*(X^i, W)}{\partial W} + \frac{\lambda}{P}\frac{\partial H(W)}{\partial W}\right]$$

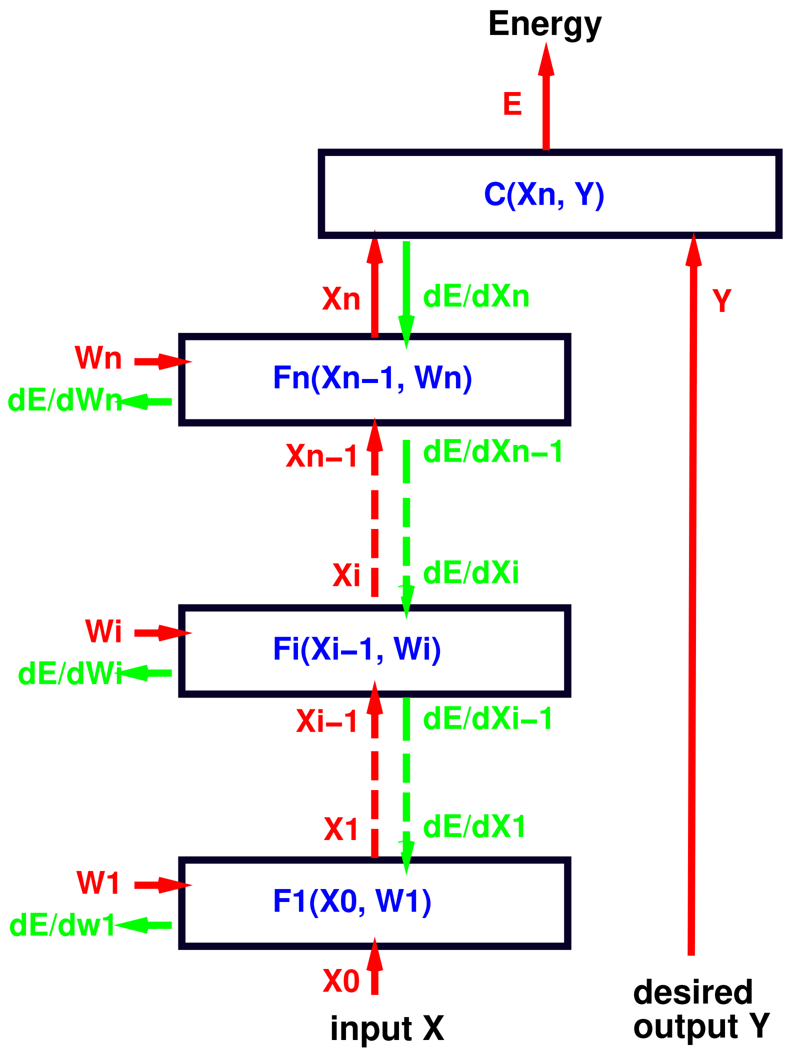
- Question: How do we compute $\frac{\partial E(Y^i, X^i, W)}{\partial W}$ efficiently?

Computing the Gradients in Multi-Layer Systems



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

Computing the Gradients in Multi-Layer Systems



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

$$\frac{\partial E}{\partial W_i} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}$$

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

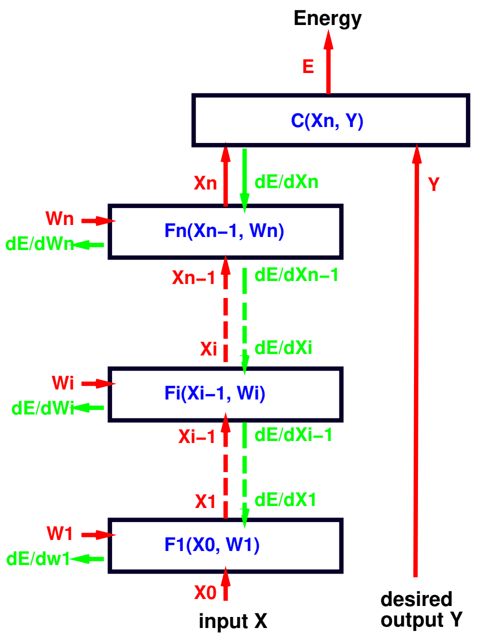
- $\frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}$ is the *Jacobian matrix* of F_i with respect to W_i.

$$\{\frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}\}_{kl} = \frac{\partial \{F_i(X_{i-1}, W_i)\}_k}{\partial \{W_i\}_l}$$

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

Computing the Gradients in Multi-Layer Systems

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



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

$$\frac{\partial E}{\partial X_{i-1}} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_{i-1}}$$

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

Jacobians and Dimensions

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

$$\frac{\partial E}{\partial X_{i-1}} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_{i-1}}$$

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

$$\frac{\partial E}{\partial W_i} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial W}$$

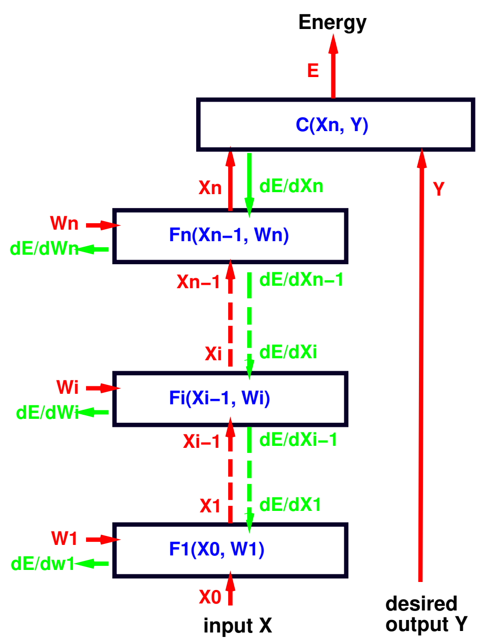
- we may prefer to write those equation with column vectors:

$$\frac{\partial E}{\partial X_{i-1}}' = \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_{i-1}}' \frac{\partial E}{\partial X_i}'$$

$$\frac{\partial E}{\partial W_i}' = \frac{\partial F_i(X_{i-1}, W_i)}{\partial W}' \frac{\partial E}{\partial X_i}'$$

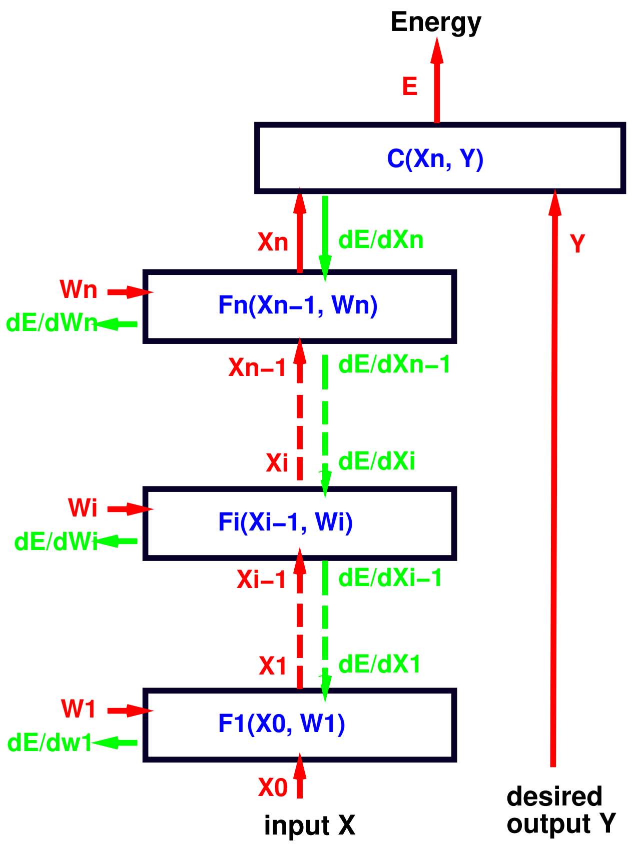
Back-propagation

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



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

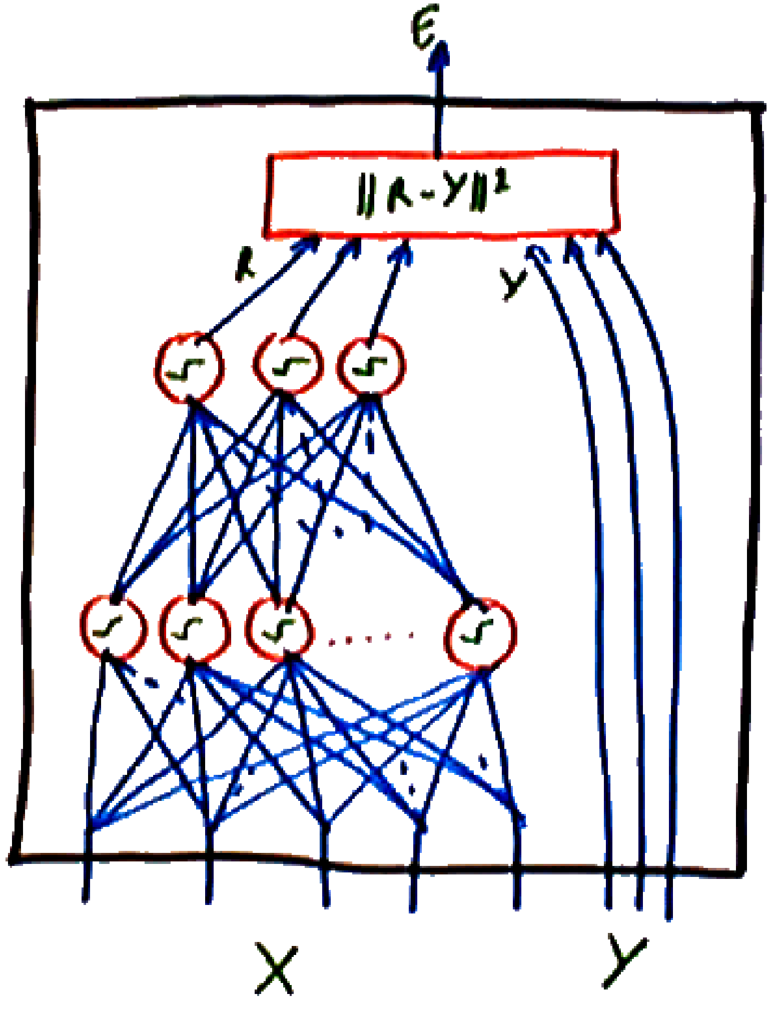
Object-Oriented Implementation



- Each module is an object (instance of a class).
- Each class has a "bprop" (backward propagation) method that takes the input and output states as arguments and computes the derivative of the energy with respect to the input from the derivative with respect to the output:
- Lush: `(==> module bprop input output)`
- C++: `module.bprop(input,output);`
- the objects `input` and `output` contain two slots: one vector for the forward state, and one vector for the backward derivatives.
- the method `bprop` computes the backward derivative slot of `input`, by multiplying the backward derivative slot of `output` by the Jacobian of the module at the forward state of `input`.

Architecture: Multi-layer Neural Network

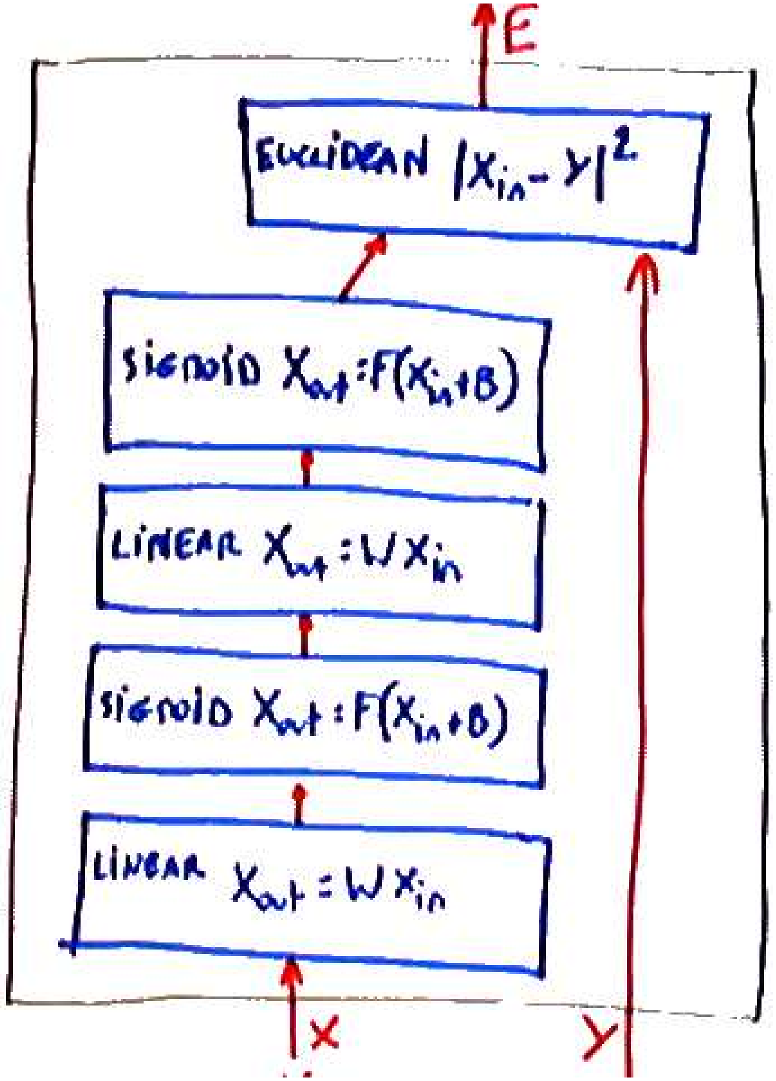
Multi-layer neural nets can be seen as networks of logistic regressors.



- Each layer is composed of a number of units (sometimes abusively called "neurons").
- Each unit performs a linear combination of its inputs, and pass the result through a sigmoid function. The result is passed on to other units.
- In a fully connected feed-forward net, the units are organized in layers.
- Each unit in one layer gets inputs from every unit in the previous layer.
- All the layers but the last are called "hidden" layers, because their state is not directly constrained from the outside (nor provided to the outside).

Modules in a Multi-layer Neural Net

A fully-connected, feed-forward, multi-layer neural nets can be implemented by stacking three types of modules.



- Linear modules: X_{in} and X_{out} are vectors, and W is a weight matrix.

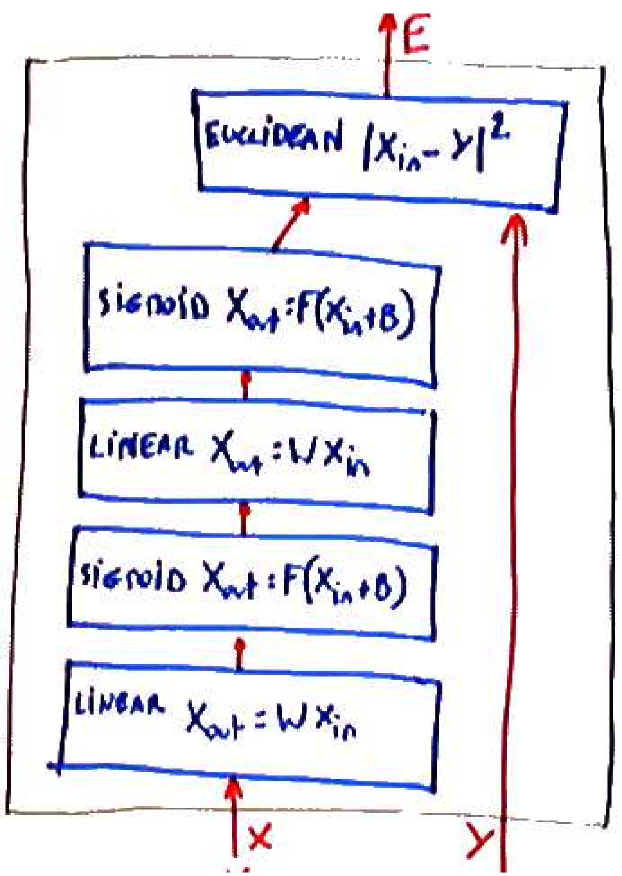
$$X_{\text{out}} = W X_{\text{in}}$$

- Sigmoid modules: $(X_{\text{out}})_i = \sigma((X_{\text{in}})_i + B_i)$ where B is a vector of trainable "biases", and σ is a sigmoid function such as $\tanh$ or the logistic function.
- a Euclidean Distance module $E = \frac{1}{2}||Y - X_{\text{in}}||^2$. With this energy function, we will use the neural network as a regressor rather than a classifier.

Objective Function

Objective function:

$$L(W) = \frac{1}{P}\sum_{i=1}^{P}[E(Y^i, X^i, W) - \frac{1}{\beta}\log\int \exp(-\beta E(Y, X^i, W))dY] + \frac{\lambda}{P}H(W)$$

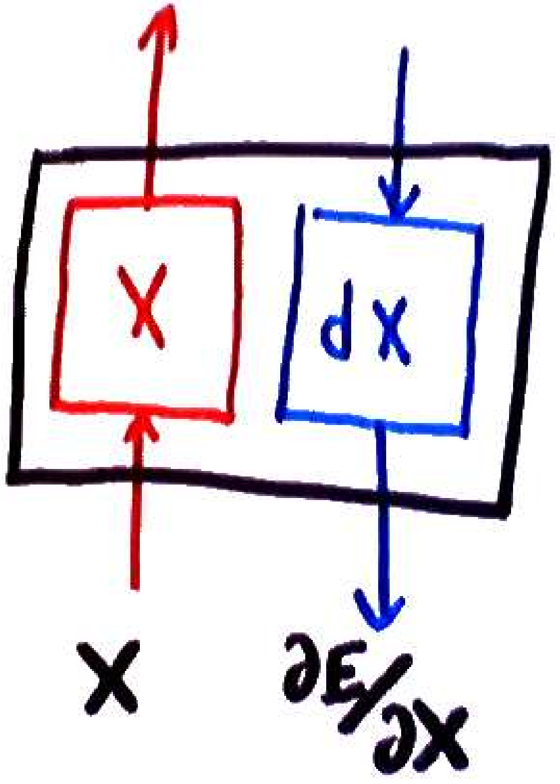


- As we have seen with linear regression, the E^* term (the integral) does not depend on W and can be happily ignored.
- The objective function is:

$$L(W) = \frac{1}{P}\sum_{i=1}^{P} E(Y^i, X^i, W) + \frac{\lambda}{P}H(W)$$

OO Implementation: the `state1` Class

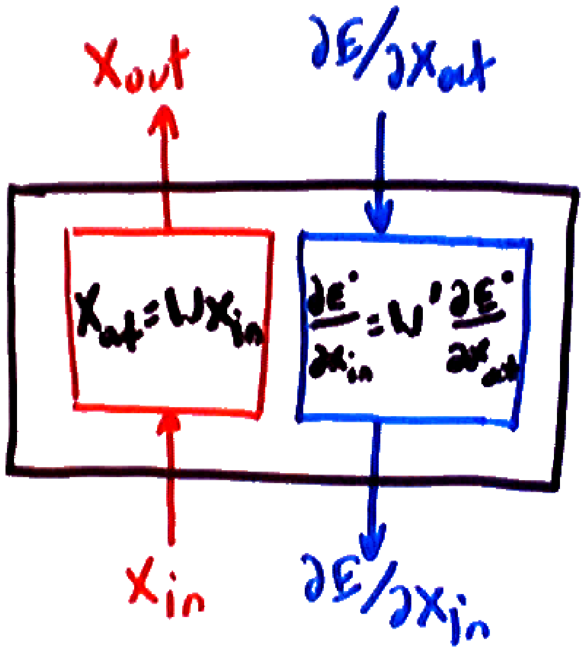
the internal state of the network will be kept in a "state" class that contains two scalars, vectors, or matrices: (1) the state proper, (2) the derivative of the energy with respect to that state.




```
#? * state
;; a state is a class that carries variables between
;; trainable modules. States can be scalars, vectors,
;; or matrices of any dimension.
(defclass state object x dx)

#? (new state [<n1> [<n2 [<n3> ...]]])
;; create a new state. The arguments are the dimensions.
(defmethod state state l
  (setq x (apply matrix l))
  (setq dx (apply matrix l)))

#? (==> <state> resize [<n1> [<n2 [<n3> ...]]])
;; resize an existing state.
(defmethod state resize l
  (idx-redim x l)
  (idx-redim dx l))
```

Linear Module

The input vector is multiplied by the weight matrix.



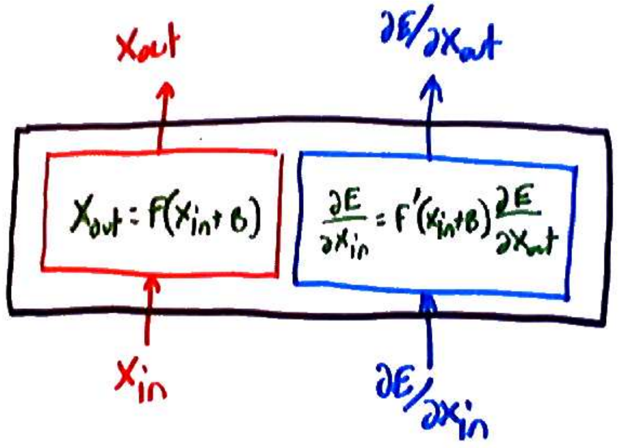
- fprop: $X_{\text{out}} = W X_{\text{in}}$
- bprop to input: $\frac{\partial E}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} \frac{\partial X_{\text{out}}}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} W$
- by transposing, we get column vectors:
  $\frac{\partial E}{\partial X_{\text{in}}}' = W' \frac{\partial E}{\partial X_{\text{out}}}'$
- bprop to weights:
  $\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial X_{\text{out}i}} \frac{\partial X_{\text{out}i}}{\partial W_{ij}} = X_{\text{in}j} \frac{\partial E}{\partial X_{\text{out}i}}$
- We can write this as an outer-product:
  $\frac{\partial E}{\partial W}' = \frac{\partial E}{\partial X_{\text{out}}}' X'_{in}$

Linear Module

Lush implementation:

```
(defclass linear-module object w)

(defmethod linear-module linear-module (ninputs noutputs)
  (setq w (new state noutputs ninputs)))

(defmethod linear-module fprop (input output)
  (idx-m2dotm1 :w:x :input:x :output:x) ())

(defmethod linear-module bprop (input output)
  (idx-m2dotm1 (transpose :w:x) :output:dx :input:dx)
  (idx-m1extm1 :output:dx :input:x :w:dx) ())
```

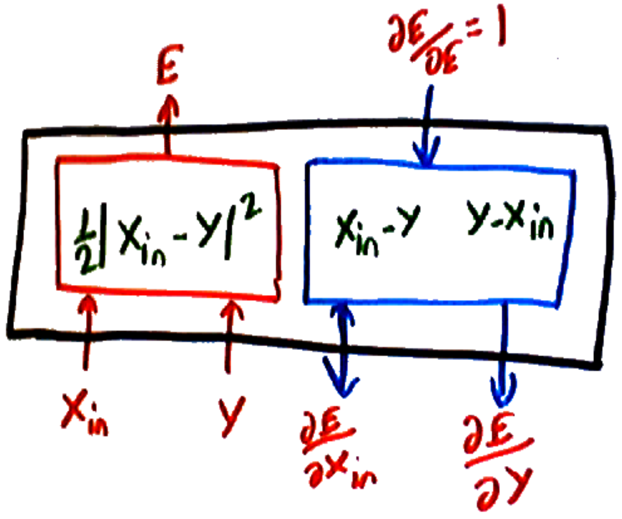
Sigmoid Module (tanh: hyperbolic tangent)



- fprop: $(X_{\text{out}})_i = \tanh((X_{\text{in}})_i + B_i)$
- bprop to input:
  $(\frac{\partial E}{\partial X_{\text{in}}})_i = (\frac{\partial E}{\partial X_{\text{out}}})_i \tanh'((X_{\text{in}})_i + B_i)$
- bprop to bias:
  $\frac{\partial E}{\partial B_i} = (\frac{\partial E}{\partial X_{\text{out}}})_i \tanh'((X_{\text{in}})_i + B_i)$
- $\tanh(x) = \frac{2}{1+\exp -x} - 1 = \frac{1-\exp(-x)}{1+\exp(-x)}$

```
(defmethod sigmoid-module object bias)
(defmethod sigmoid-module sigmoid-module (n)
  (setq bias (new state n)))
(defmethod sigmoid-module fprop (input output)
  (idx-add :input:x :bias:x :output:x)
  (idx-qtanh :output:x :output:x))
(defmethod sigmoid-module bprop (input output)
  (idx-qdtanh (idx-add :input:x :bias:x) :input:dx)
  (idx-mul :input:dx :output:dx :input:dx)
  (idx-copy :input:dx :bias:dx) ())
```

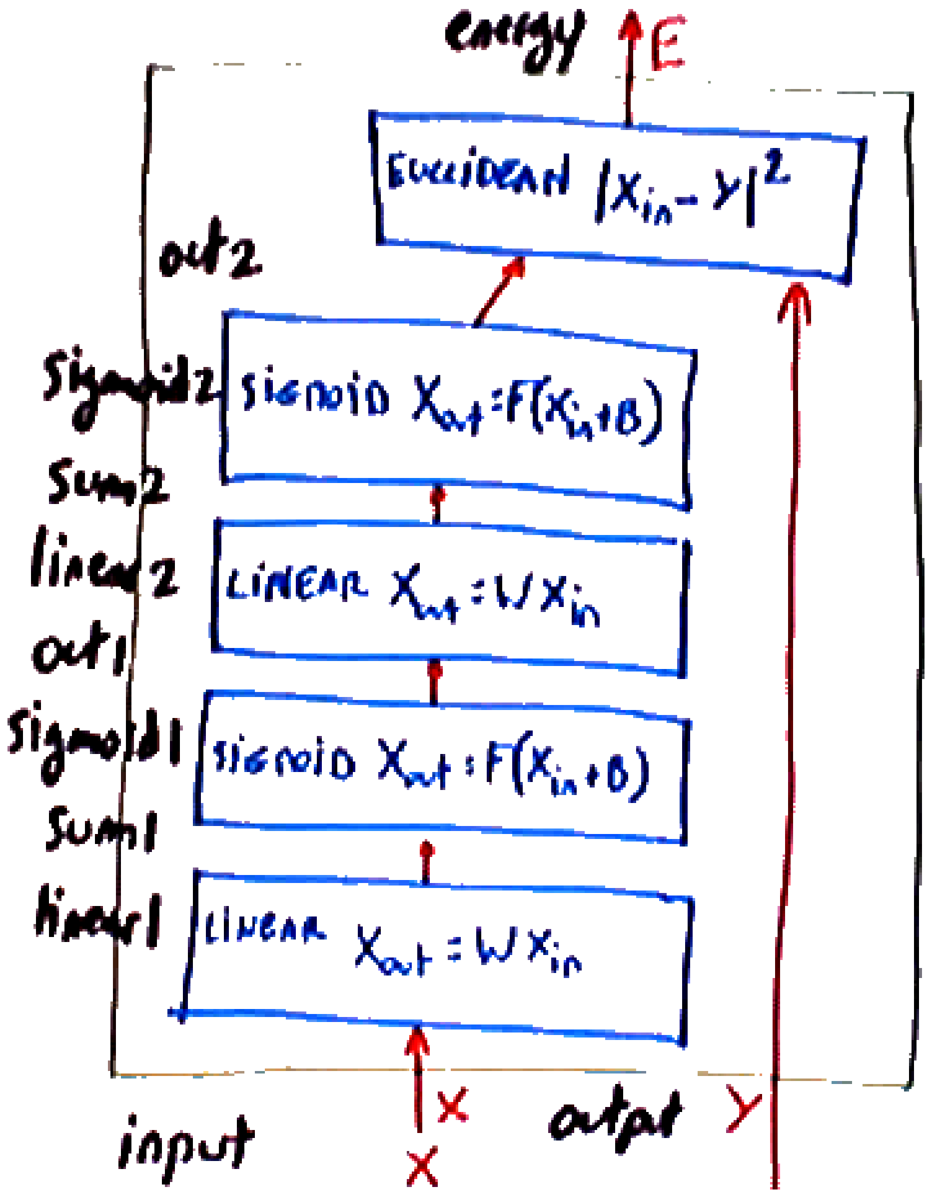
Euclidean Module



- fprop: $X_{\text{out}} = \frac{1}{2}||X_{\text{in}} - Y||^2$
- bprop to X input: $\frac{\partial E}{\partial X_{\text{in}}} = X_{\text{in}} - Y$
- bprop to Y input: $\frac{\partial E}{\partial Y} = Y - X_{\text{in}}$

```
(defclass euclidean-module object)
(defmethod euclidean-module fprop (input1 input2 output)
  (idx-sqrdist :input1:x :input2:x :output:x)
  (:output:x (* 0.5 :output:x)) ())
(defmethod euclidean-module bprop (input1 input2 output)
  (idx-sub :input1:x :input2:x :input1:dx)
  (idx-dotm0 :input1:dx :output:dx :input1:dx)
  (idx-minus :input1:dx :input2:dx))
```

Assembling the Network: class definition



- Class implementation for a 2 layer, feed forward neural net.

```
;; class definition
(defclass neural-net-2 object
  linear1 sum1 sigmoid1 out1
  linear2 sum2 sigmoid2 out2
  euclidean)
```

Assembling the Network: constructor

Constructor:

```
#? (new neural-net-2 <ninput> <nhidden> <noutput>)
;; constructor
(defmethod neural-net-2 neural-net-2 (ninput nhidden noutput)
    (setq linear1 (new linear-module ninput nhidden))
    (setq sum1 (new state1 nhidden))
    (setq sigmoid1 (new sigmoid-module nhidden))
    (setq out1 (new state1 nhidden))
    (setq linear2 (new linear-module nhidden noutput))
    (setq sum2 (new state1 noutput))
    (setq sigmoid2 (new sigmoid-module noutput))
    (setq out2 (new state1 noutput))
    (setq euclidean (new euclidean-module)) ())
```

Assembling the Network: run and fprop

Implementation of a 2 layer, feed forward neural net.

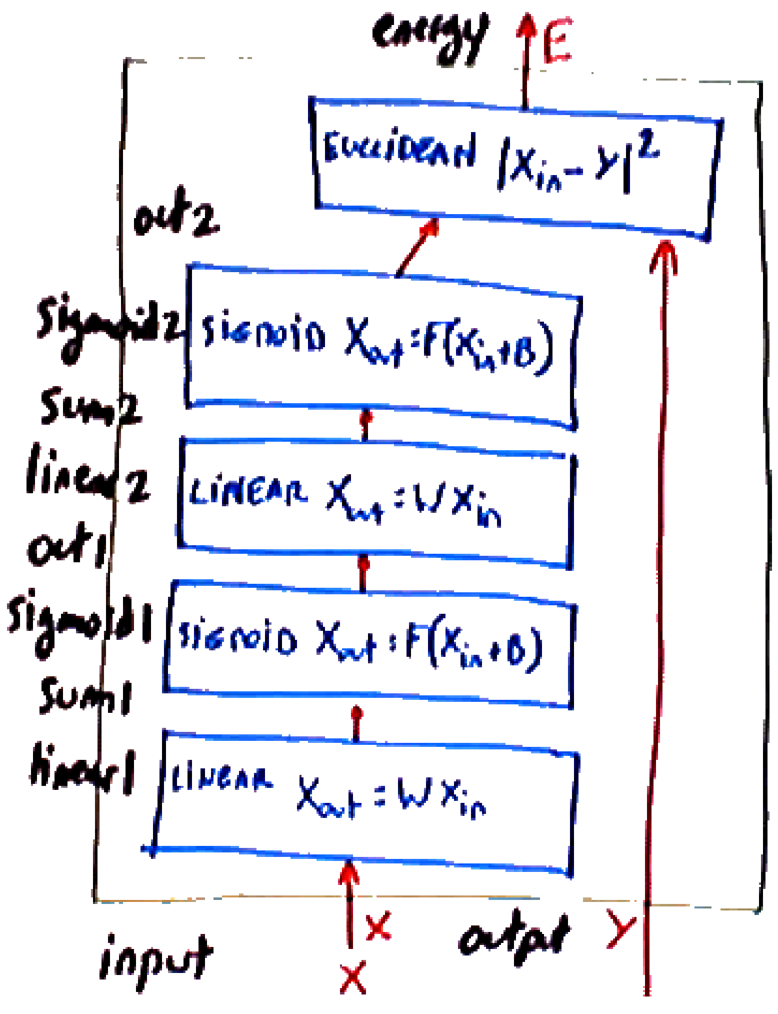
```
;; run method: find the output that minimizes the energy
(defmethod neural-net-2 run (input output energy)
  (==> linear1 fprop input sum1)
  (==> sigmoid1 fprop sum1 out1)
  (==> linear2 fprop out1 sum2)
  (==> sigmoid2 fprop sum2 out2)
  (idx-copy :out2:x :output:x)
  (:energy:x 0) ())

;; fprop method: compute energy for a given X and Y.
(defmethod neural-net-2 fprop (input desired energy)
  (==> linear1 fprop input sum1)
  (==> sigmoid1 fprop sum1 out1)
  (==> linear2 fprop out1 sum2)
  (==> sigmoid2 fprop sum2 out2)
  (==> euclidean fprop out2 desired energy) ())
```

Assembling the Network: bprop

Class implementation for a 2 layer, feed forward neural net.

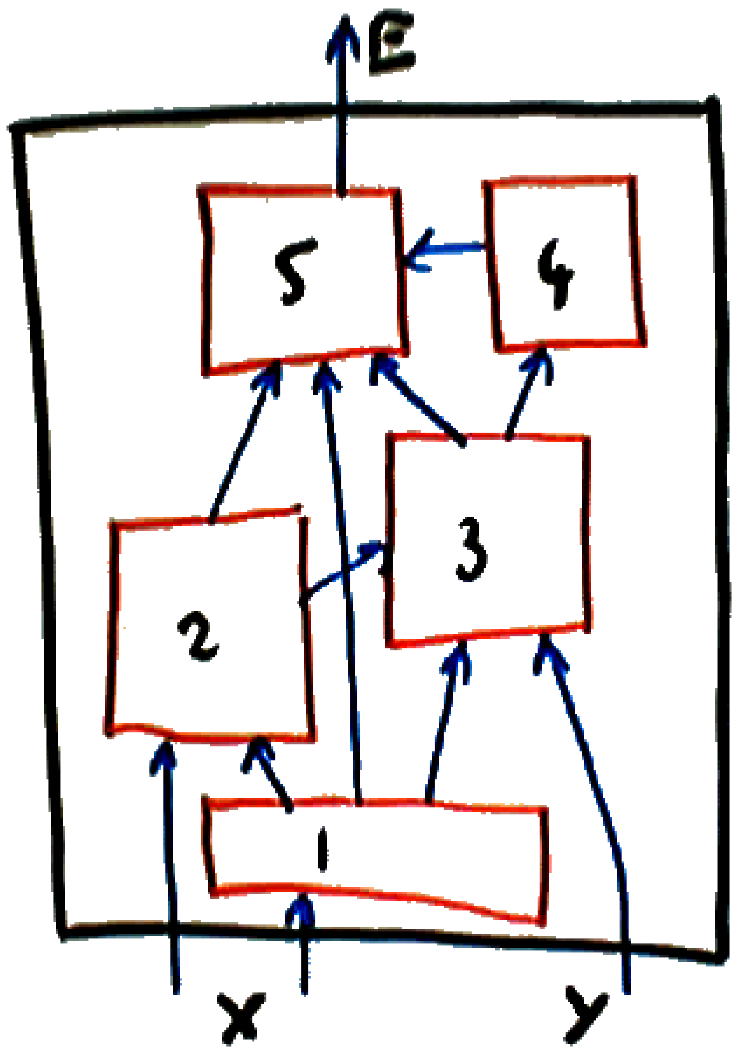
```
;; bprop method: compute all the gradients.
;; This assumes that the dx slot of the energy
;; is properly initialized to 1.
(defmethod neural-net-2 bprop (input desired energy)
  (==> euclidean bprop out2 desired energy)
  (==> sigmoid2 bprop sum2 out2)
  (==> linear2 bprop out1 sum2)
  (==> sigmoid1 bprop sum1 out1)
  (==> linear1 bprop input sum1) ())
```

Assembling the Network: training



- A training cycle:
- pick a sample (X^i, Y^i) from the training set.
- call fprop with (X^i, Y^i) and record the error
- call bprop with (X^i, Y^i)
- update all the weights using the gradients obtained above.
- with the implementation above, we would have to go through each and every module to update all the weights. In the future, we will see how to "pool" all the weights and other free parameters in a single vector so they can all be updated at once.

Other Topologies



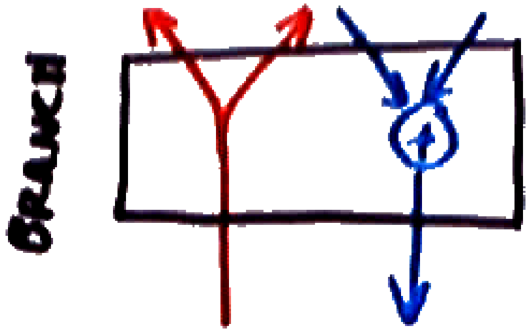
- The back-propagation procedure is not limited to feed-forward cascades.
- It can be applied to networks of module with *any* topology, as long as the connection graph is acyclic.
- If the graph is acyclic (no loops) then, we can easily find a suitable order in which to call the fprop method of each 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.

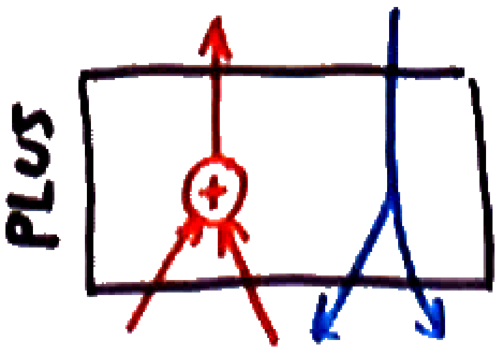
More Modules

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:

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

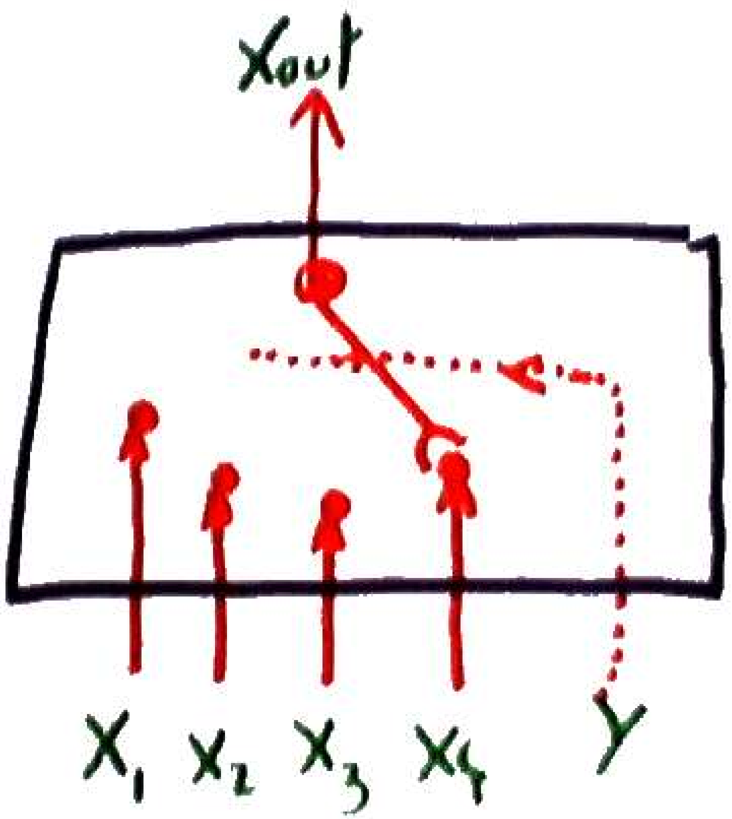
back-prop: $\frac{\partial E}{\partial X_k} = \frac{\partial E}{\partial X_{\mathrm{out}}} \quad \forall k$

- 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:

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

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

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

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

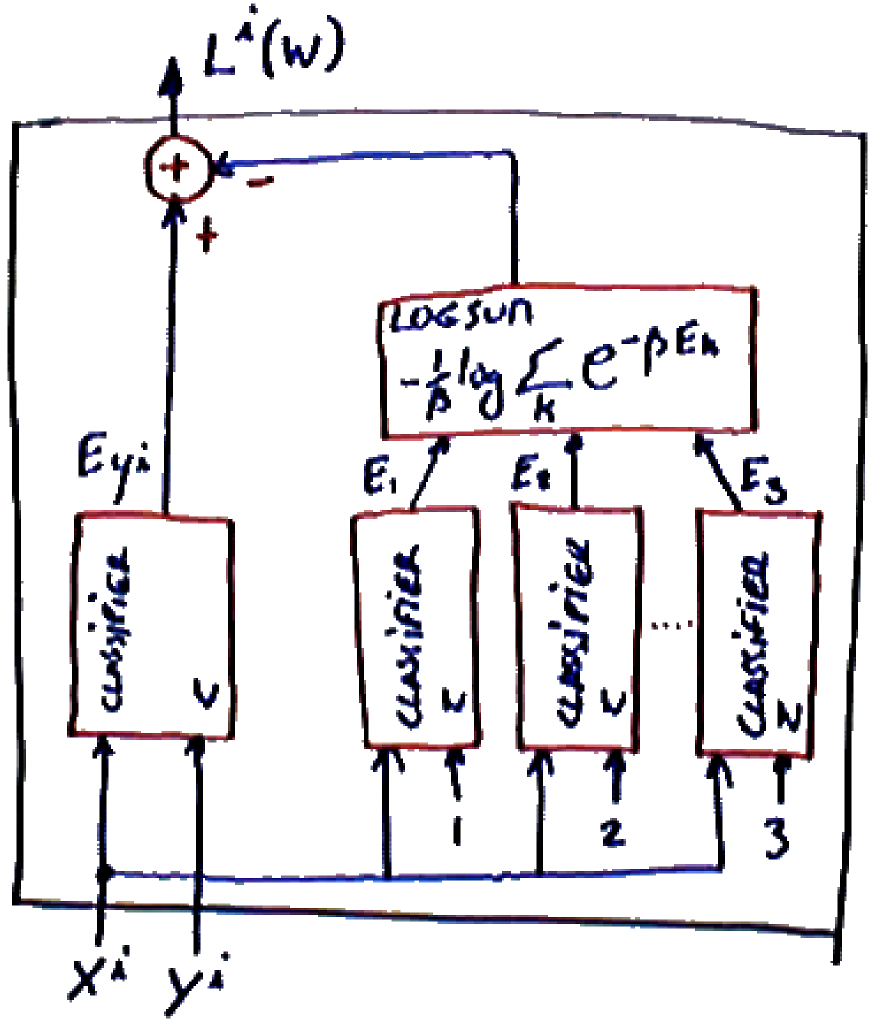
with

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

Loss function and Logsum Modules

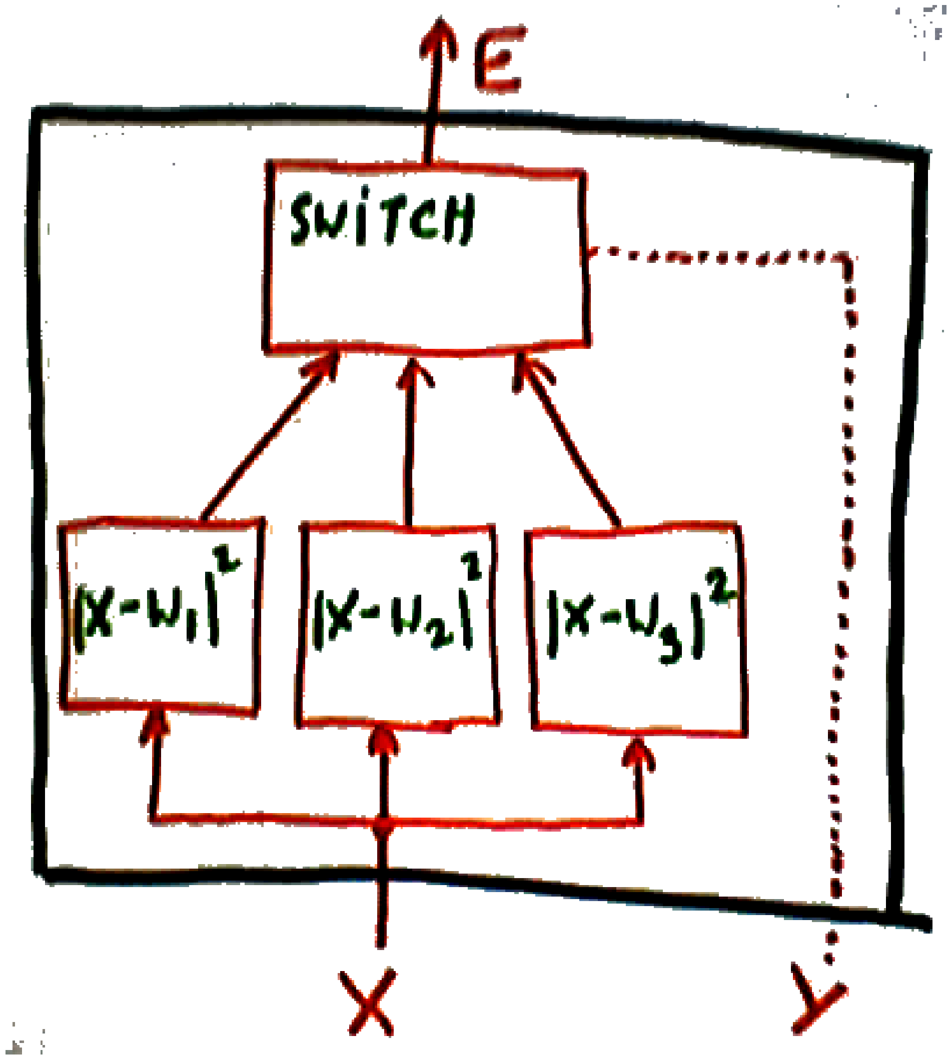
MAP/MLE Loss

$L(W) = \frac{1}{P}\sum_{i=1}^{P}[E(Y^i, X^i, W) + \frac{1}{\beta}\log\sum_k \exp(-\beta E(k, X^i, W))]$



- This loss function can be computed by building a machine with multiple "replicas" of the classifier, one replica for the desired output, and K replicas for each possible value of Y.

Switch-based Classifier



- K modules that measure the dissimilarity of the incoming pattern each of K categories (e.g. Euclidean modules).
- Recognition mode: find the position of the switch (value of Y) that minimizes the energy. In other words, find the module among the K whose W vector best matches the input.
- Probabilistic recognition mode:

$$P(Y|X) = \frac{\exp(-\beta|X - W_Y|^2)}{\sum_k \exp(-\beta|X - W_k|^2)}$$

Switch-based Classifier

Learning mode: The loss function is:

$$L(W) = \frac{1}{P}\sum_{i=1}^{P}[E(Y^i, X^i, W) + \frac{1}{\beta}\log\int \exp(-\beta E(Y, X^i, W))dY] + \frac{\lambda}{P}H(W)$$

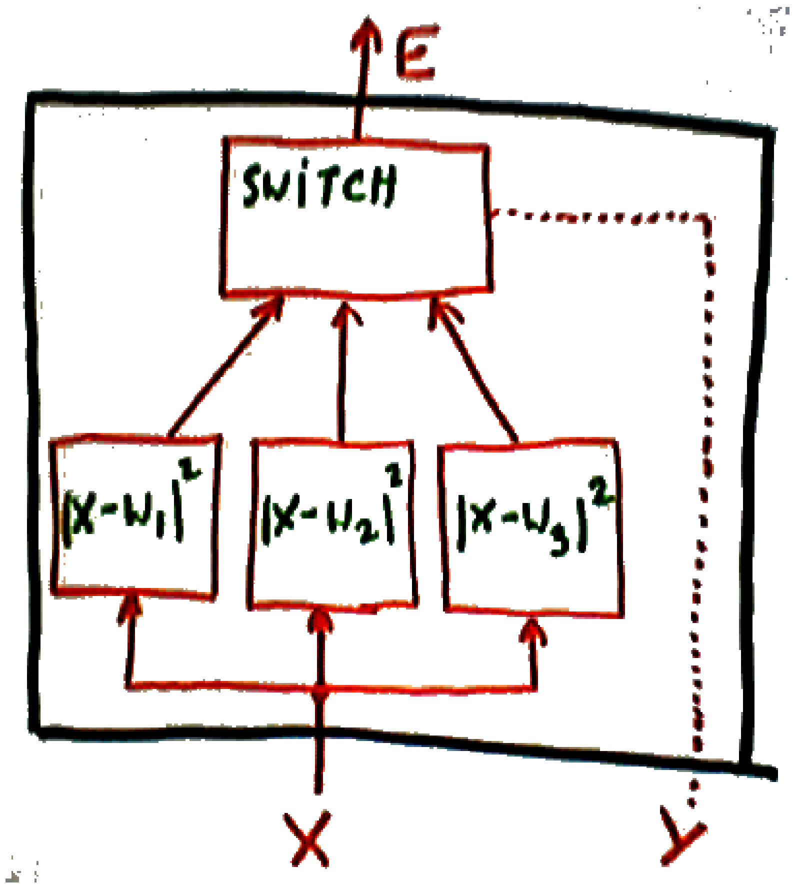
If the discrete input to the switch is the label Y, the term

$$\frac{1}{\beta}\log\int \exp(-\beta E(Y, X^i, W))dY$$

becomes a discrete sum: $\frac{1}{\beta}\log\sum_k \exp(-\beta E(k, X^i, W))$ and the loss function becomes:

$$L(W) = \frac{1}{P}\sum_{i=1}^{P}[E(Y^i, X^i, W) + \frac{1}{\beta}\log\sum_k \exp(-\beta E(k, X^i, W))]$$

More on Switch-based Classifiers

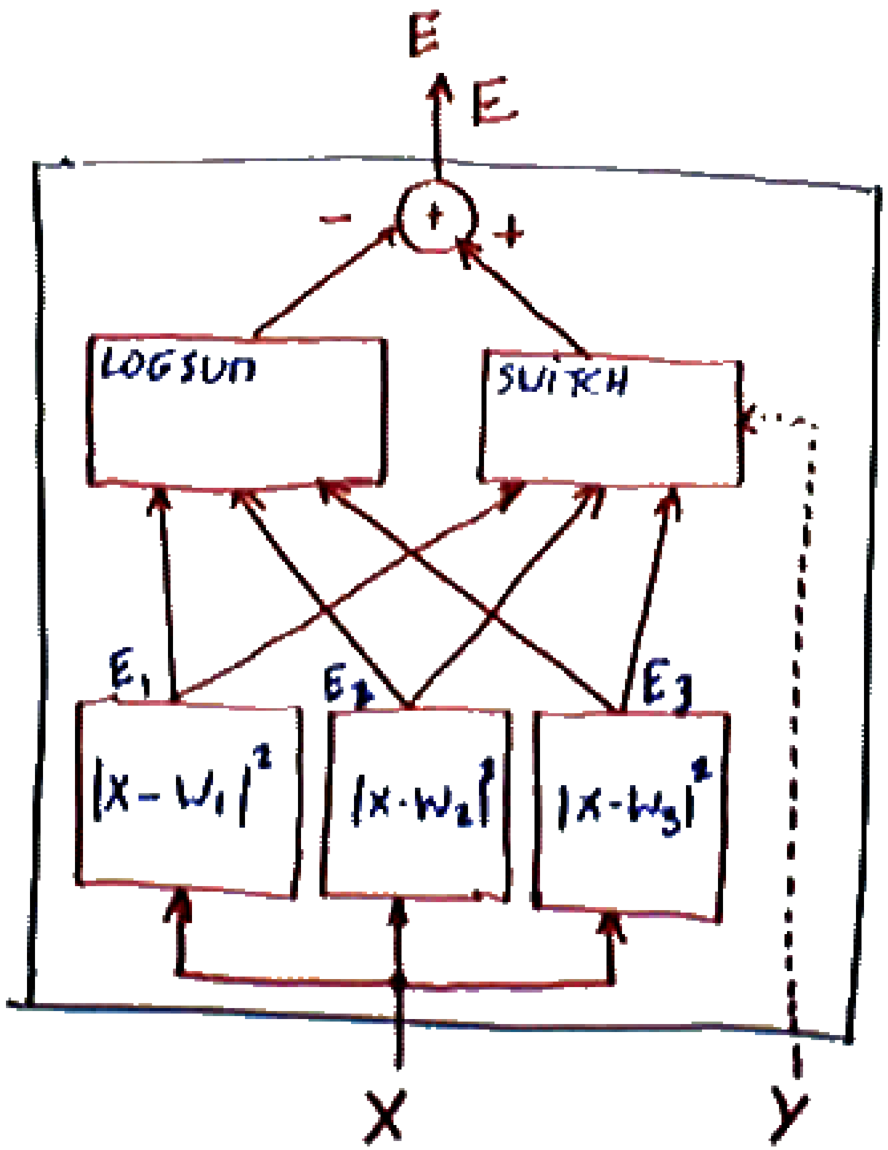


- $L(W) = \frac{1}{P}\sum_{i=1}^{P}[E(Y^i, X^i, W) + \frac{1}{\beta}\log\sum_k \exp(-\beta E(k, X^i, W))]$
- If the switch is the last module in the chain, let's denote by E_k it's k-th input.
- The loss function for one example (X^i, Y^i) becomes:

$$L^i(W) = E_{Y^i} + \frac{1}{\beta}\log\sum_k \exp(-\beta E_k)$$

- Using a switch as the last module in the chain with the traditional MAP/MLE loss function is equivalent to adding a *logsum* module whose output is subtracted from the switch output.

More on Switch-based Classifiers



- $L^i(W) =$
  $E_{Y^i} + \frac{1}{\beta} \log \sum_k \exp(-\beta E_k)$
- The switch-based classifier with the MAP/MLE loss function is equivalent to this "collapsed" classifier.

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

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