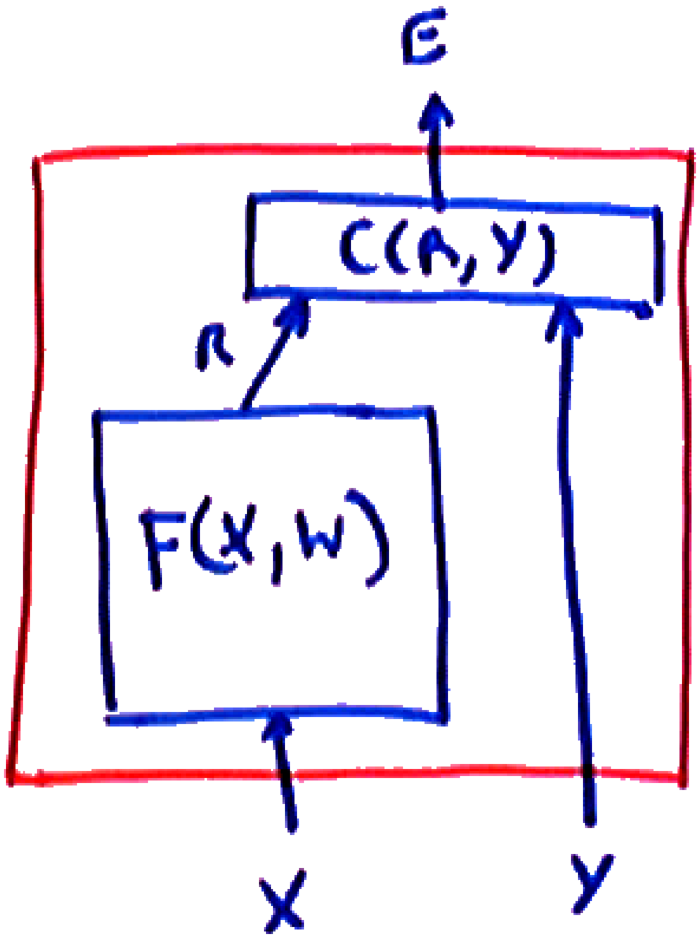
MACHINE LEARNING AND PATTERN RECOGNITION

Spring 2004, Lecture 4a

Back-Propagation, Multilayer and Multi-Module Systems.

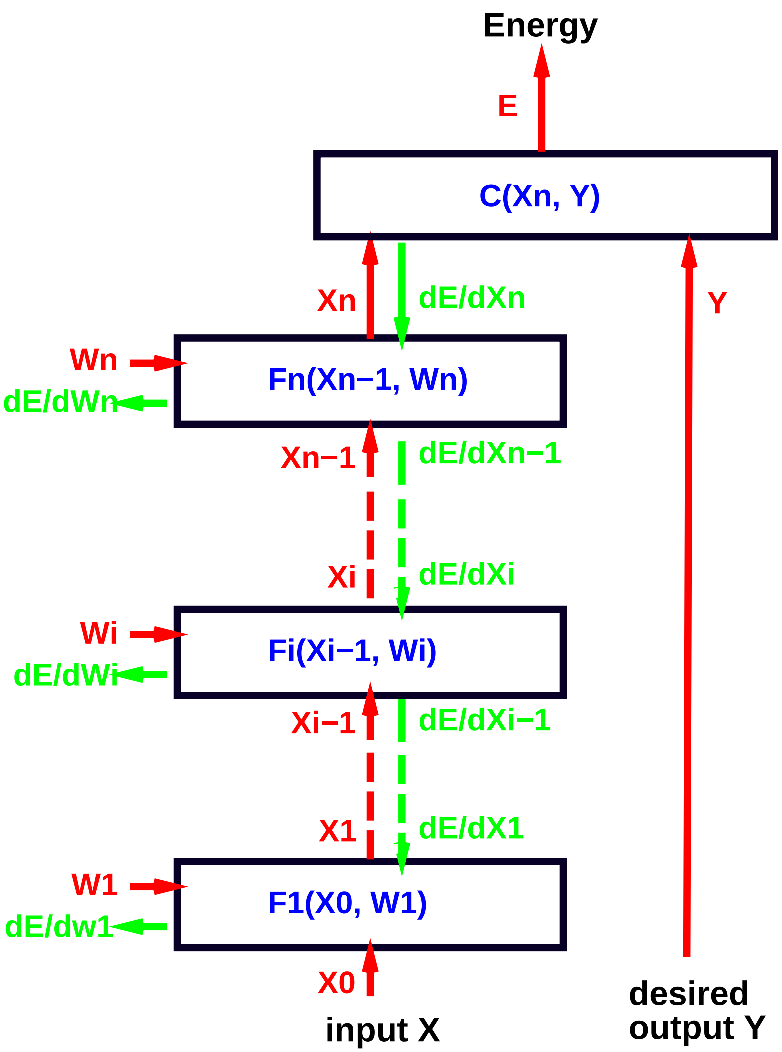
Yann LeCun
The Courant Institute,
New York University
http://yann.lecun.com

Non-Linear Learning



- So far, we have seen how to train linear machines, and we have hinted at the fact that we could also train non-linear machines.
- In non-linear machines, 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: Cascade

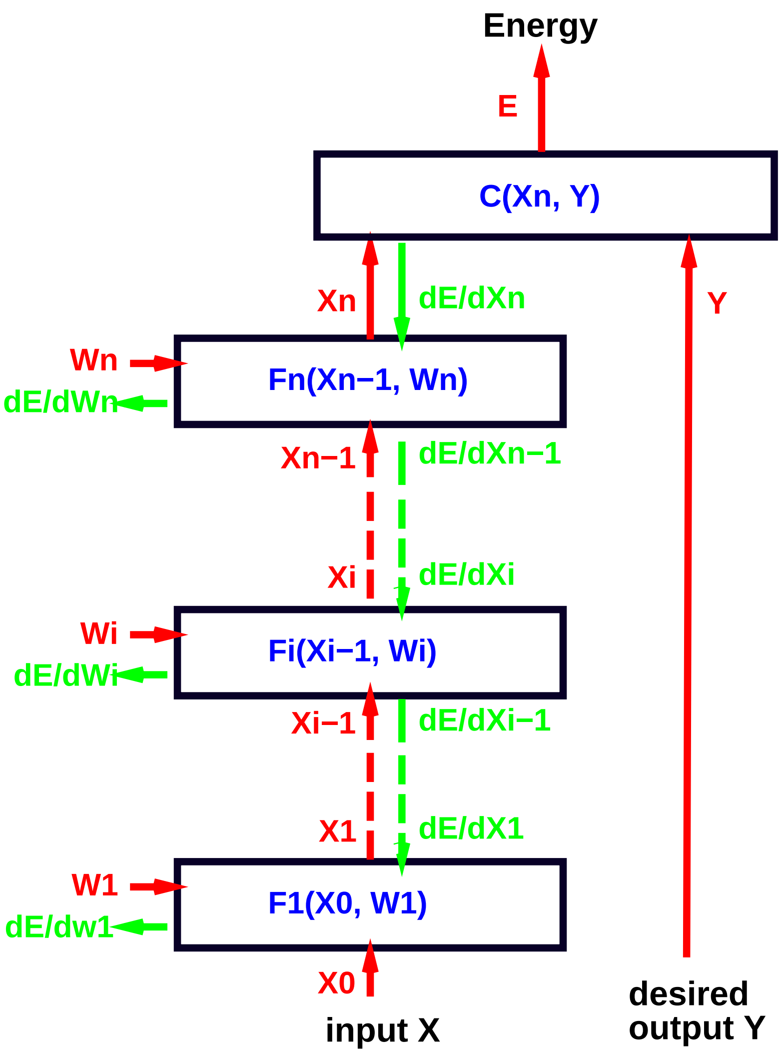


- Complex learning machines can be built by assembling Modules into networks.
- a simple example: layered, feed-forward architecture (cascade).
- computing the output from the input: forward propagation
- let $X = X_0$,

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

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

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 average over a training set $\{(X^1, Y^1), (X^2, Y^2), \dots, (X^P, Y^P)\}$ of a loss function such as:

$$L_{\text{energy}}(W, Y^i, X^i) = E(W, Y^i, X^i)$$

$$L_{\text{perceptron}}(W, Y^i, X^i) = \left[E(W, Y^i, X^i) - \min_y E^*(W, y, X^i) \right] + \lambda H(W)$$

$$L_{\text{ll}}(W, Y^i, X^i) = \left[E(W, Y^i, X^i) + \frac{1}{\beta} \log \int \exp(-\beta E(W, y, X^i)) dy \right] + \lambda H(W)$$

- λ is an appropriately picked coefficient that determines the importance of the regularization term.

Gradient of the Loss, gradient of the Energy

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

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

  $L(W, Y^i, X^i)$ depends on X^i only through the $E(W, Y, X^i)$ for all Y, so we can apply chain rule (see next page):

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

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

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

Gradient of the Loss, gradient of the Energy

- We assumed early on that the loss depends on W only through the terms $E(W, Y, X^i)$:

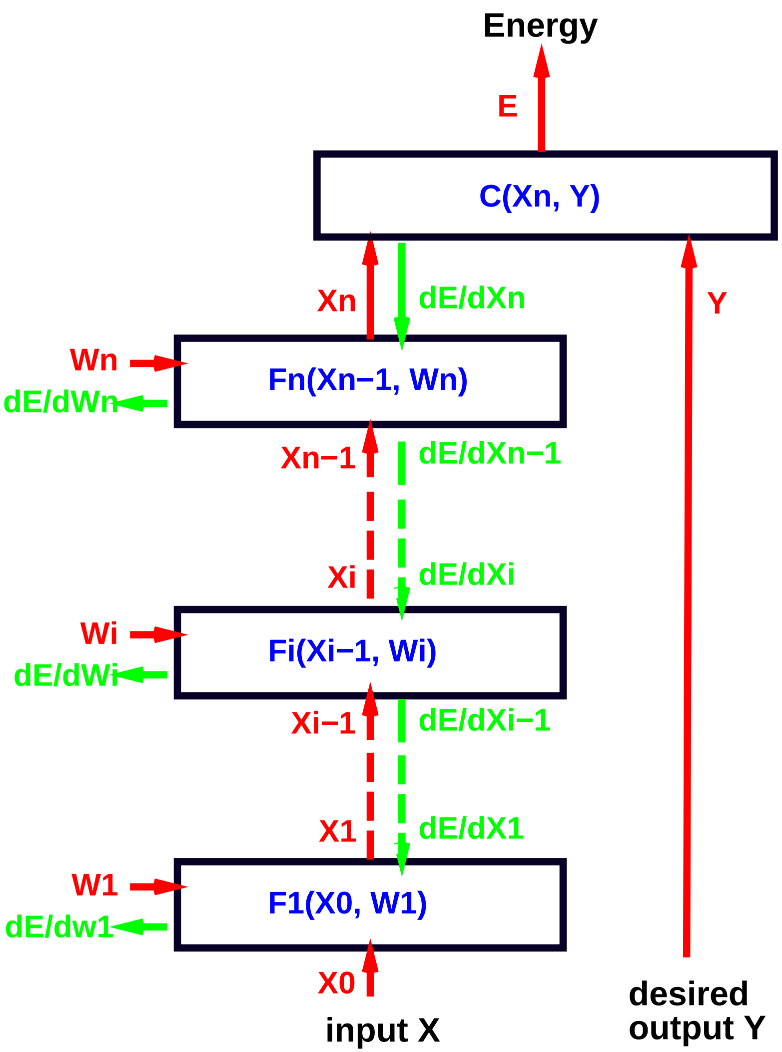
$$L(W, Y^i, X^i) = L(Y^i, E(W, 0, X^i), E(W, 1, X^i), \ldots, E(W, k-1, X^i))$$

- therefore:

$$\frac{\partial L(W, Y^i, X^i)}{\partial W} = \sum_Y \frac{\partial L(W, Y^i, X^i)}{\partial E(W, Y, X^i)} \frac{\partial E(W, Y, X^i)}{\partial W}]$$

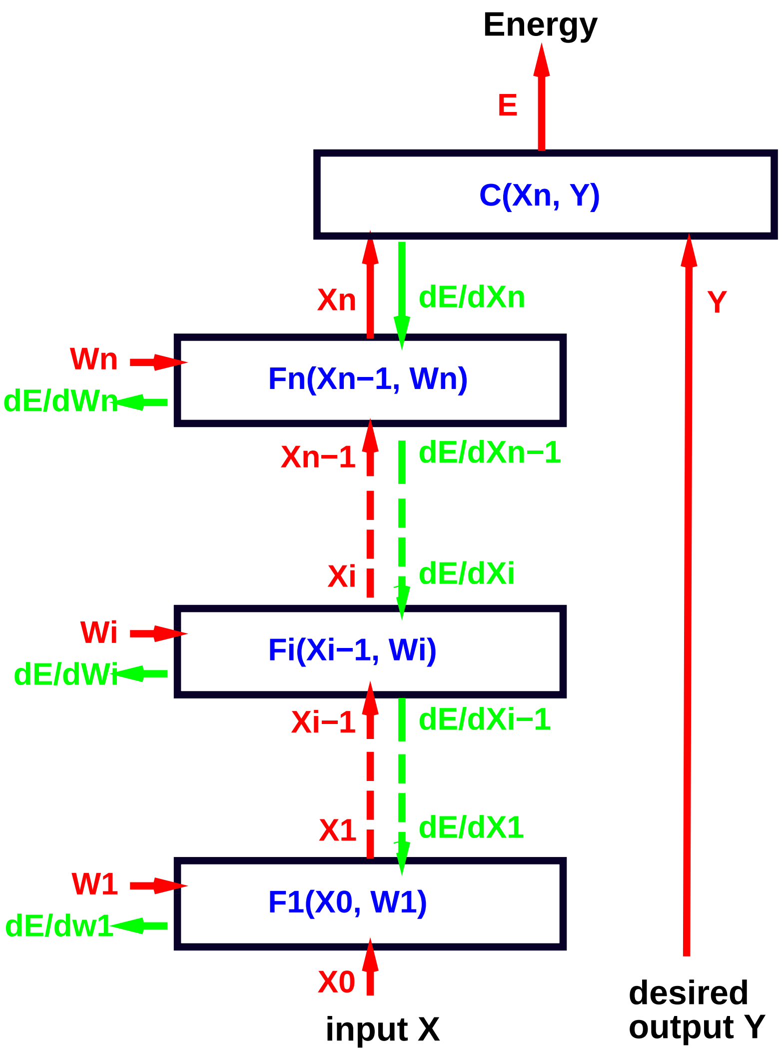
- We only need to compute the terms $\frac{\partial E(W,Y,X^i)}{\partial W}$
- Question: How do we compute those terms efficiently?

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):

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

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

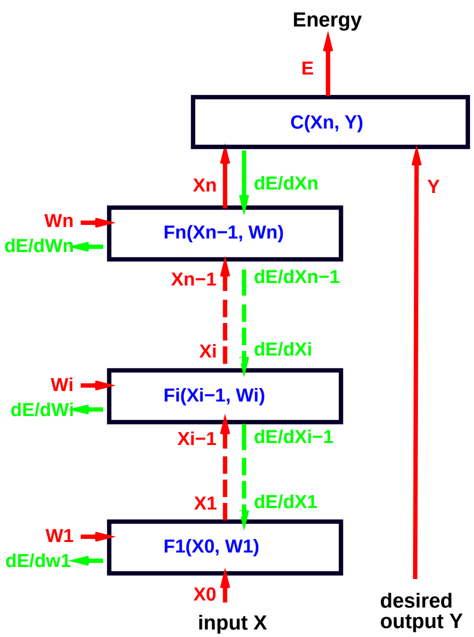
- $\frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}$ is the *Jacobian matrix* of F_i with respect to W_i.

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

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

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

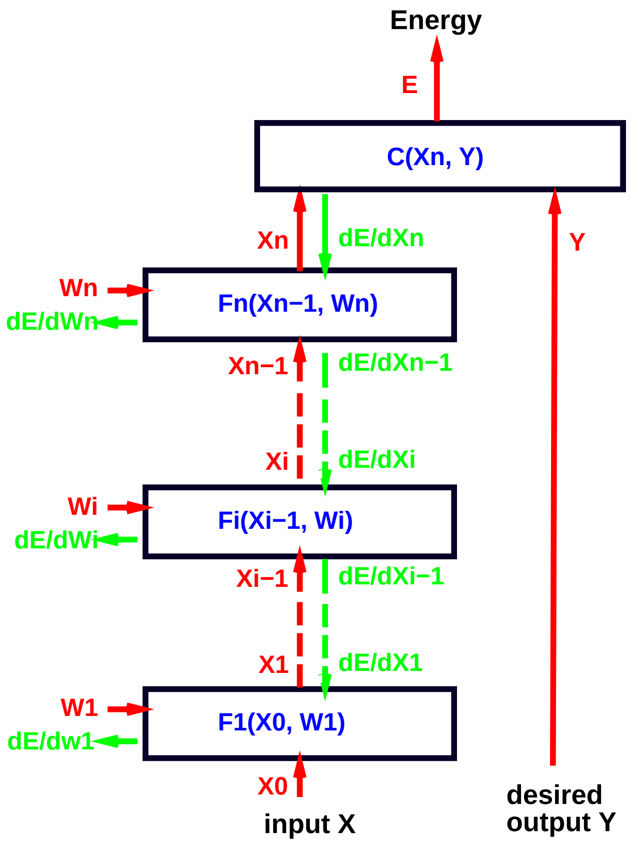
- we may prefer to write those equation with column vectors:

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

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

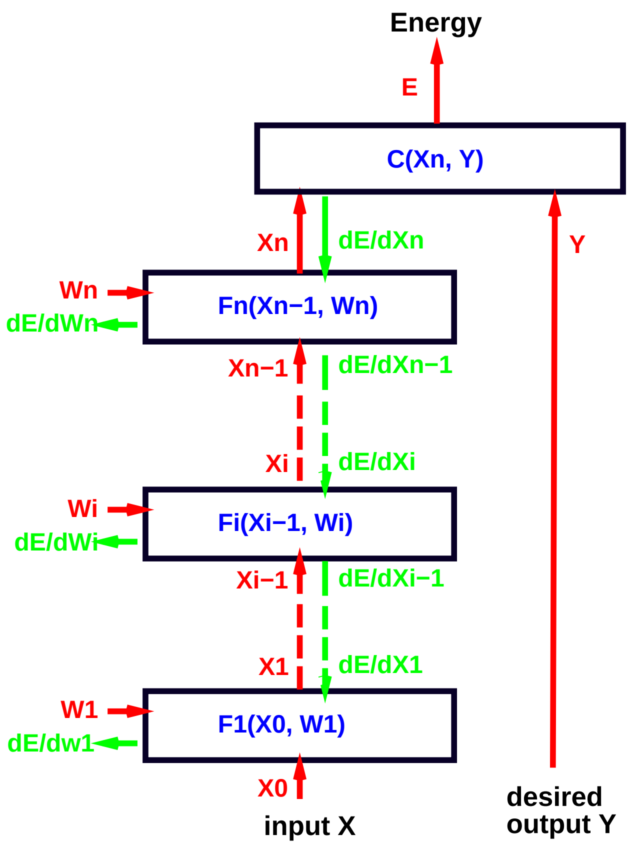
Back-propagation

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 $i \in [1, n]$.

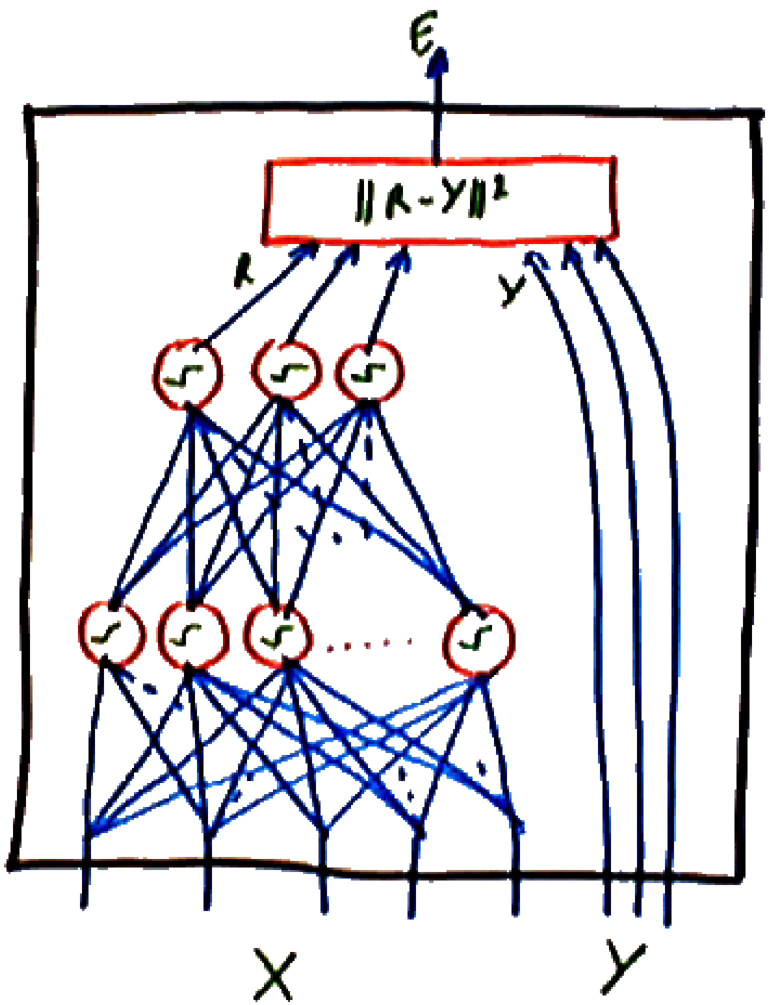
Object-Oriented Implementation



- 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

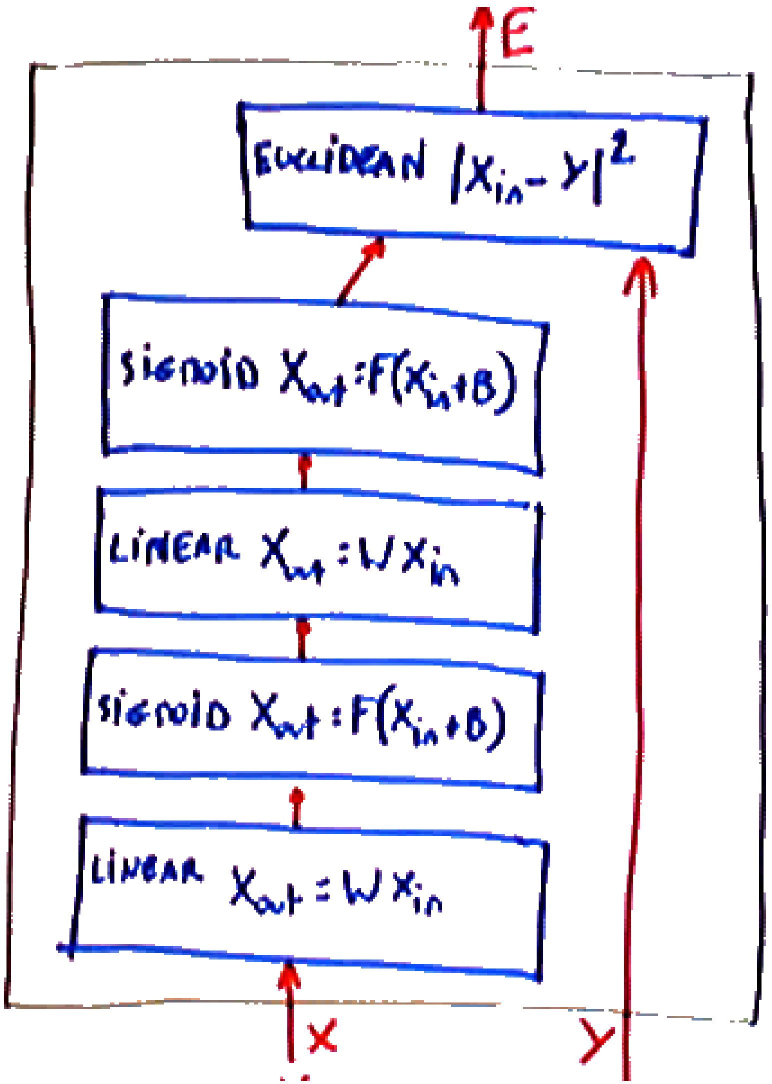
Multi-layer neural nets can be seen as networks of logistic regressors.



- 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.

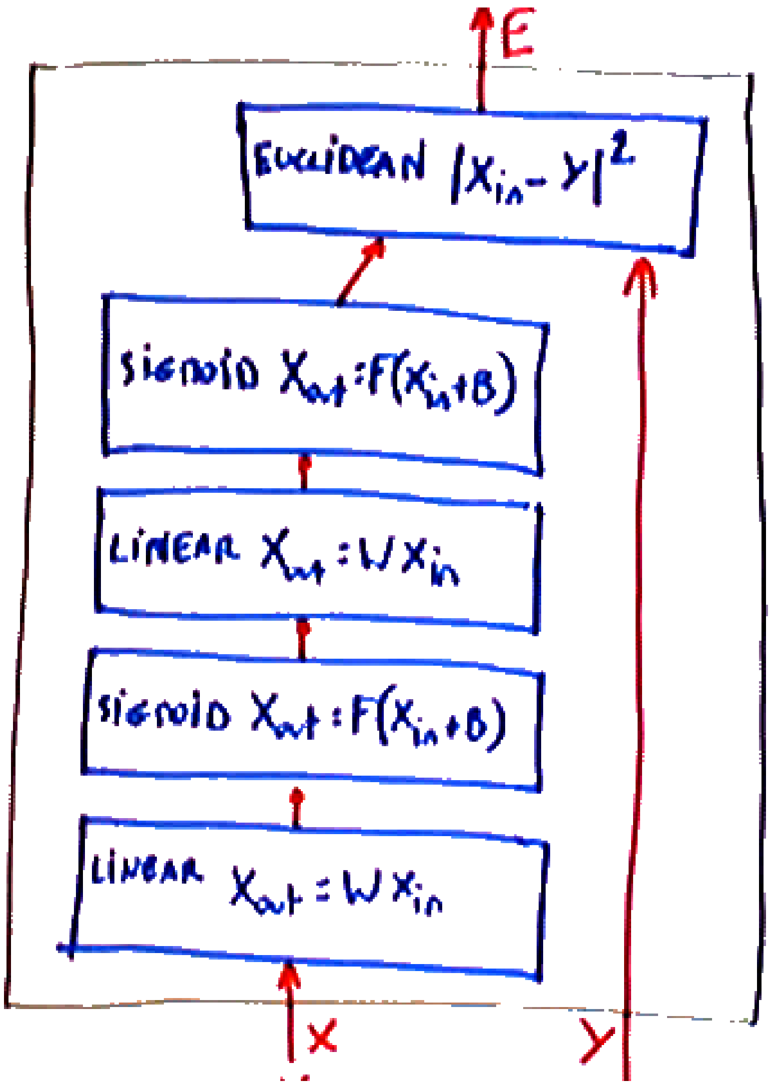


- Linear modules: X_{in} and X_{out} are vectors, and W is a weight matrix.

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

- 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.

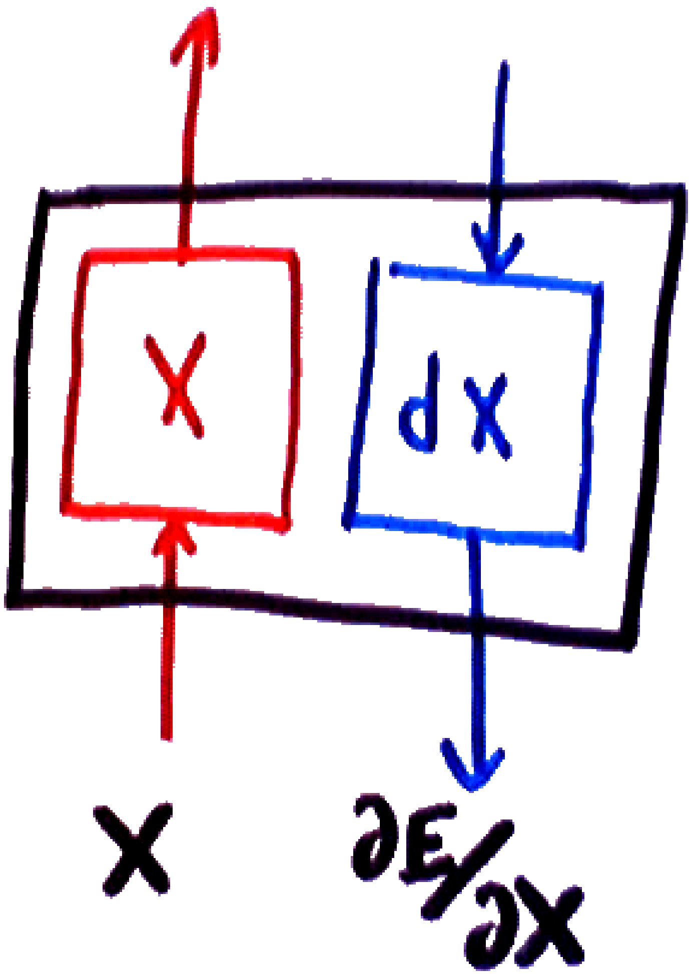
Loss Function



Here, we will us the simple Energy Loss function L_{energy}:

$$L_{\text{energy}}(W, Y^i, X^i) = E(W, Y^i, X^i)$$

OO Implementation: the `state` 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.

OO Implementation: the `state1` Class

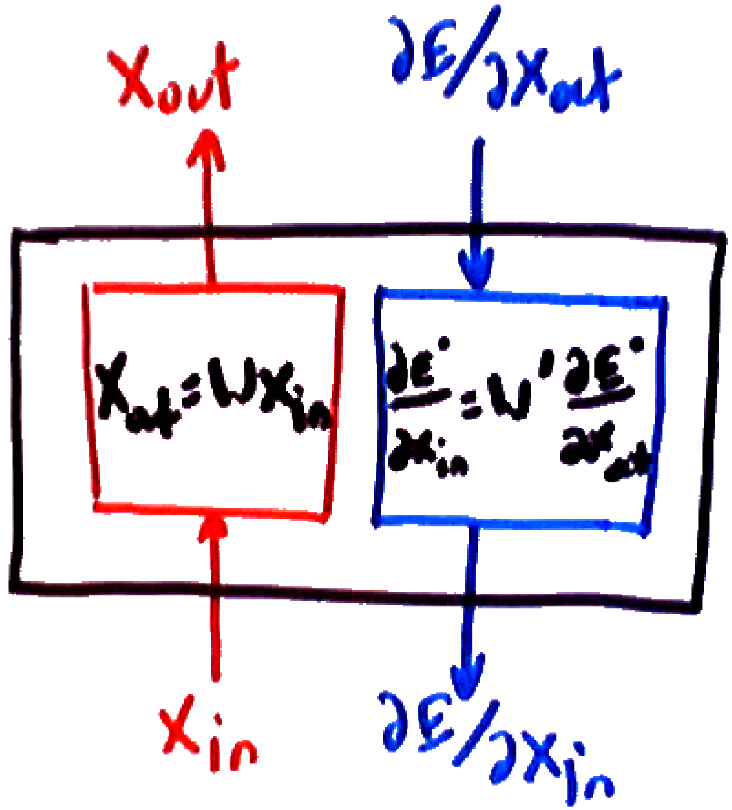
```
#? * state
;; a <state> is a class that carries variables between
;; trainable modules. States can be scalars, vectors,
;; matrices, tensors of any dimension, or any other
;; type of objects. A state contains a slot <x> to contain
;; the actual state, and a slot <dx> to contain the
;; partial derivatives of the loss function with respect
;; to the state variables.
(defclass state object x dx)

#? (new state [<n1> [<n2 [<n3> ...]]])
;; create a new state. The arguments
;; are the dimensions (up to 8 dimensions).
(defmethod state state l
  (setq x (apply matrix l))
  (setq dx (apply matrix l)))

#? (==> <state> resize [<n1> [<n2 [<n3> ...]]])
;; resize an existing state the the dimensions
;; passed as arguments.
(defmethod state resize l
```

Linear Module

The input vector is multiplied by the weight matrix.



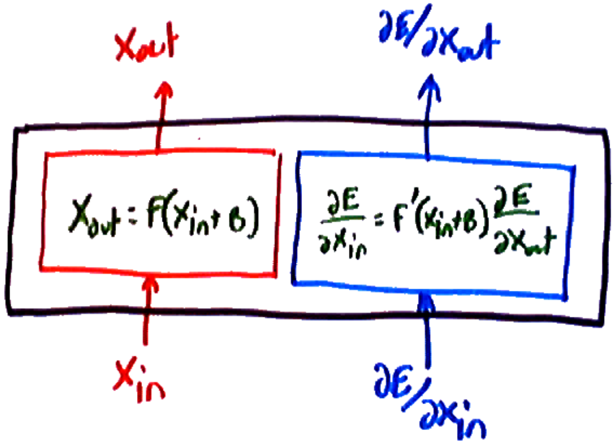
- fprop: $X_{\text{out}} = W X_{\text{in}}$
- bprop to input:
  $\frac{\partial E}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} \frac{\partial X_{\text{out}}}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} W$
- by transposing, we get column vectors:
  $\frac{\partial E}{\partial X_{\text{in}}}' = W' \frac{\partial E}{\partial X_{\text{out}}}'$
- 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 (matrix noutputs ninputs)))
(defmethod linear-module fprop (input output)
  (==> output resize (idx-dim :w:x 0))
  (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) ())
```

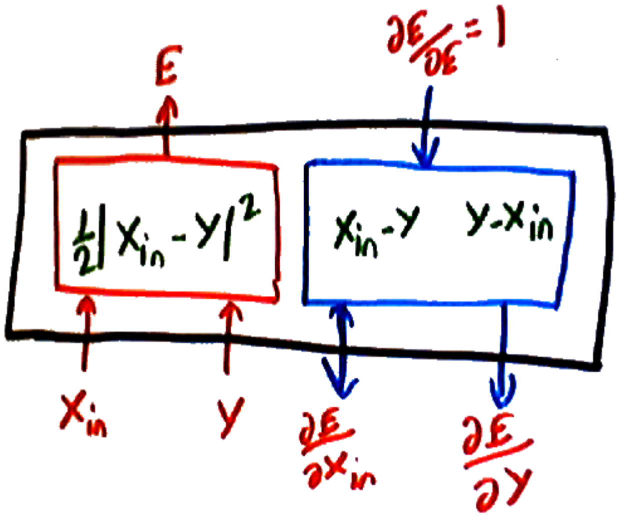
Sigmoid Module (tanh: hyperbolic tangent)



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

```
(defclass tanh-module object bias)
(defmethod tanh-module tanh-module l
  (setq bias (apply matrix l)))
(defmethod tanh-module fprop (input output)
  (==> output resize (idx-dim :bias:x 0))
  (idx-add :input:x :bias:x :output:x)
  (idx-tanh :output:x :output:x))
(defmethod tanh-module bprop (input output)
  (idx-dtanh (idx-add :input:x :bias:x) :input:dx)
  (idx-mul :input:dx :output:dx :input:dx)
  (idx-copy :input:dx :bias:dx) ())
```

Euclidean Module



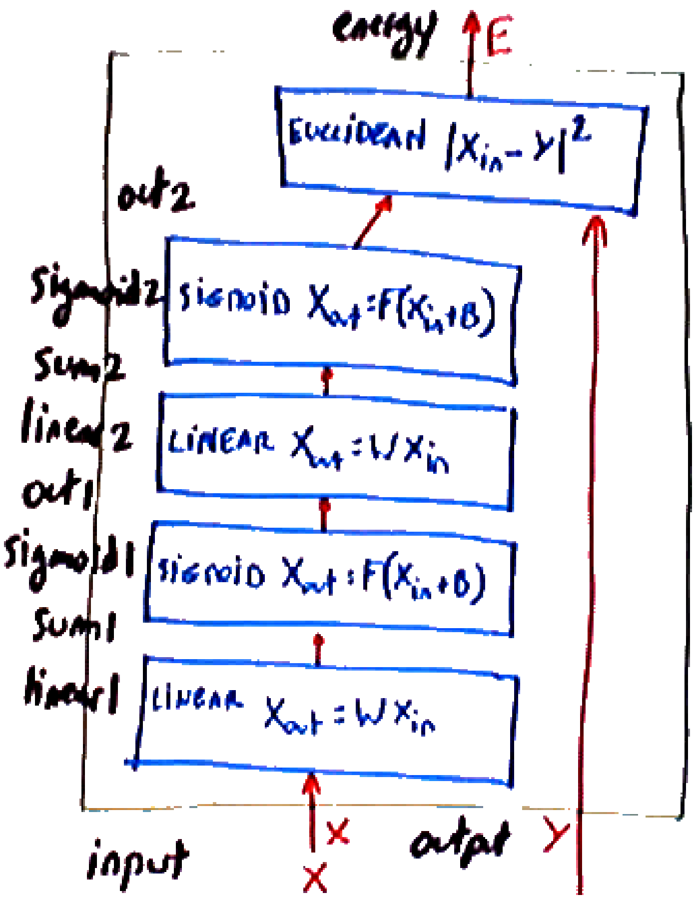
- fprop: $X_{\text{out}} = \frac{1}{2}||X_{\text{in}} - Y||^2$
- bprop to X input: $\frac{\partial E}{\partial X_{\text{in}}} = X_{\text{in}} - Y$
- bprop to Y input: $\frac{\partial E}{\partial Y} = Y - X_{\text{in}}$

```
(defclass euclidean-module object)
(defmethod euclidean-module run (input1 input2 output)
  (idx-copy :input1:x :input2:x)
  (:output:x 0) ())
(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: A single layer

```
;; One layer of a neural net
(defclass nn-layer object
  linear  ; linear module
  sum  ; weighted sums
  sigmoid  ; tanh-module
  )
(defmethod nn-layer nn-layer (ninputs noutputs)
  (setq linear (new linear-module ninputs noutputs))
  (setq sum (new state noutputs))
  (setq sigmoid (new tanh-module noutputs)) ())
(defmethod nn-layer fprop (input output)
  (==> linear fprop input sum)
  (==> sigmoid fprop sum output) ())
(defmethod nn-layer bprop (input output)
  (==> sigmoid bprop sum output)
  (==> linear bprop input sum) ())
```

Assembling a 2-layer Net



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

```
(defclass nn-2layer object
  layer1 ; first layer module
  hidden ; hidden state
  layer2 ; second layer
  )
(defmethod nn-2layer nn-2layer (ninputs nhid
  (setq layer1 (new nn-layer ninputs nhidden)
  (setq hidden (new state nhidden))
  (setq layer2 (new nn-layer nhidden noutputs
```

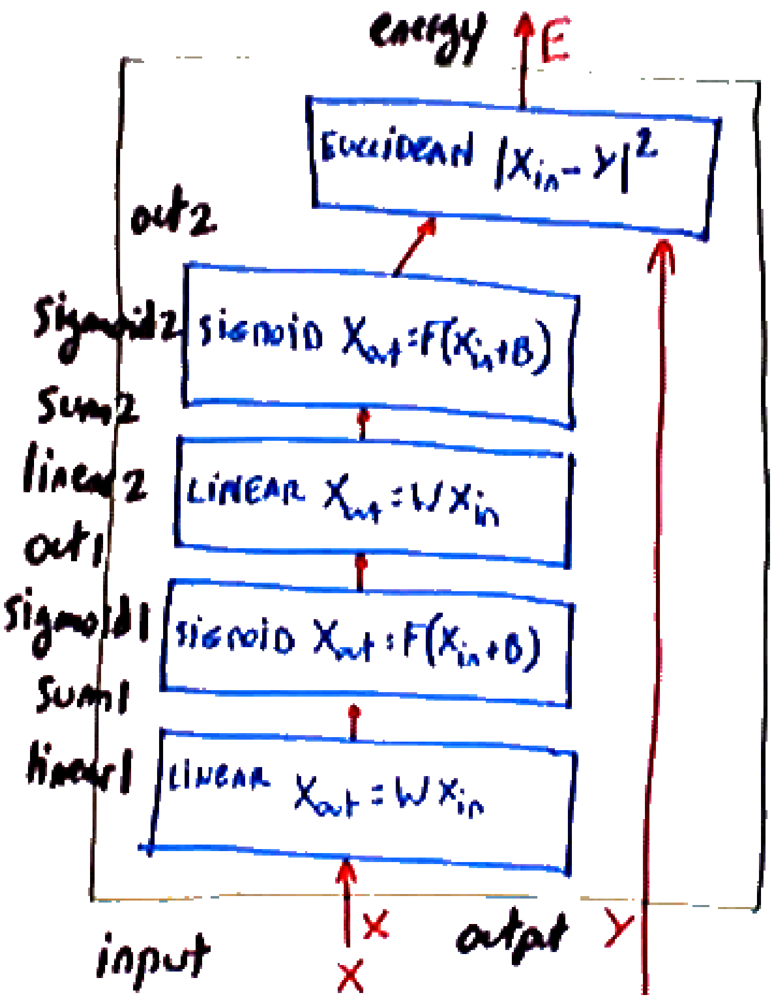
Assembling the Network: fprop and bprop

Implementation of a 2 layer, feed forward neural net.

```
(defmethod nn-2layer fprop (input output)
  (==> layer1 fprop input hidden)
  (==> layer2 fprop hidden output) ())

(defmethod nn-2layer bprop (input output)
  (==> layer2 bprop hidden output)
  (==> layer1 bprop input hidden) ())
```

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.