Linear Models

- In the two class case, the linear classifier is given by

$$h(X) = \text{sign}(W^T X + w_0)$$

- We have seen that we can also think of $h(X)$ as

$$h(X) = \text{sign}(W^T \Phi(X) + w_0),$$

$$\text{where} \quad \Phi(X) = [\phi_1(X), \cdots, \phi_m(X)]^T$$

  as long as ϕ_i are fixed (possibly non-linear) functions.
- We have seen many methods for learning linear models.

Beyond linear models

- Learning linear models (classifiers) is generally efficient.
- However, linear models are not always sufficient.
- We have looked at three broad approaches to learning nonlinear classifiers.
- We next discuss neural network models.

Neural network models

- Artificial neural networks provide a good class of parameterized nonlinear functions.
- Nonlinear functions are built up through composition of summation and sigmoids.
- Useful for both classification and Regression.

What is an Artificial Neural Network?

A parallel distributed information processor made up of simple processing units that has *a propensity for acquiring problem solving knowledge through experience*.

- Large number of inter connected units
- Each unit implements simple function, nonlinear
- The ‘knowledge’ resides in the interconnection strengths.
- Problem solving ability is often through ‘learning’

An architecture inspired by the structure of Brain

Artificial Intelligence (AI)

- 'understanding' 'intelligence' in computational terms.
- Developing 'machines' that are 'intelligent'.

At least two distinct approaches

- **Try to model intelligent behavior in terms of processing structured symbols. (Resulting methods, algorithms etc may not resemble brain at the architectural level)**
- **A second approach is based on / inspired by human brain at architectural level**

The symbolic AI approach

- Brain is to be understood in computational terms **only**
- Physical symbol system hypothesis
- A digital computer is a universal symbol manipulator and can be programmed to be intelligent
- A 'knowledge-intensive' approach. (Formal Logic and Search are the main tools)

An implicit faith: The architecture of Brain *per se* is irrelevant for engineering intelligent artifacts

Artificial Neural Networks

- Can be viewed as one approach towards understanding brain/building intelligent machines
- Computational architectures inspired by brain.
- Computational methods for 'learning from data'
- Modeling Brain?
  - Uses Mathematically purified neurons!!
  - Can still give some insights

Artificial Neural Networks

Computing machines/architectures motivated by brain architecture.

- A large network of interconnected units
- Each unit has simple input-output mapping
- Each interconnection has numerical weight attached to it
- Output of unit depends on outputs and connection weights of units connected to it
- problem solving ability is through learning

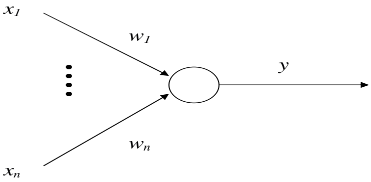
Why study such models?

- A belief that the architecture of Brain is critical to intelligent behavior.
- Models can implement highly nonlinear functions. They are adaptive and can be trained.
- Very useful in many machine learning applications
- Models can help us understand Brain function (*Computational neuroscience*)

Many different models are possible

- Evolution:
  - Discrete time / continuous time
  - synchronous / asynchronous
  - deterministic / stochastic
- Interconnections:
  - Feedforward / having feedback
- States or outputs of units:
  - binary / finitely many / continuous

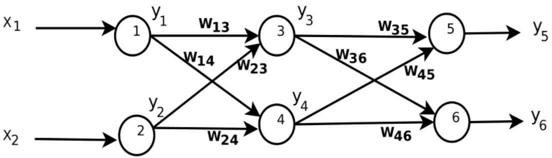
Single neuron model



- x_i are inputs into the (artificial) neuron and w_i are the corresponding *weights*. y is the output of the neuron
- Net input : $\eta = \sum_j w_j x_j$
- output: $y = f(\eta)$, where $f(.)$ is called activation function (Perceptron, AdaLinE are such models).

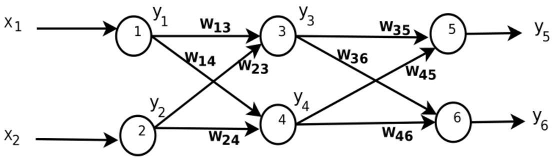
Networks of neurons

- We can connect a number of such units or neurons to form a network. Inputs to a neuron can be outputs of other neurons (and/or external inputs).



- Notation:
  y_j – output of j^{th} neuron;
  w_{ij} – weight of connection from neuron i to neuron j.

- Each neuron computes weighted sum of inputs and passes it through its activation function, to compute output



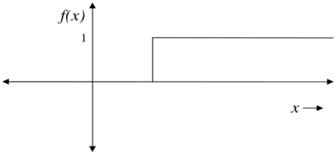
$$\begin{aligned} y_5 &= f_5\left(w_{35}\, y_3 + w_{45}\, y_4\right) \\ &= f_5\left(w_{35}\, f_3(w_{13}y_1 + w_{23}y_2) + w_{45}\, f_4(w_{14}y_1 + w_{24}y_2)\right) \end{aligned}$$

- By convention, we take $y_1 = x_1$ and $y_2 = x_2$.
- Here, x_1, x_2 are inputs and y_5, y_6 are outputs.
- Nonlinearity of activation function is important.

- A single neuron 'represents' a class of functions from $\Re^m$ to $\Re$.
- Specific set of weights realise specific functions.
- By interconnecting many units/neurons, networks can represent more complicated functions from $\Re^m$ to $\Re^{m'}$.
- The architecture constrains the function class that can be represented. Weights define specific function in the class.

Typical activation functions

1. Hard limiter:

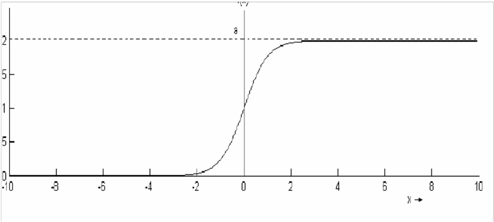


$$
\begin{aligned}
f(x) &= 1 \text{ if } x > \tau \\
&= 0 \text{ otherwise}
\end{aligned}
$$

- We can keep the τ to be zero and add one more input line to the neuron. An example of a single neuron with this activation function is Perceptron.

Typical Activation functions

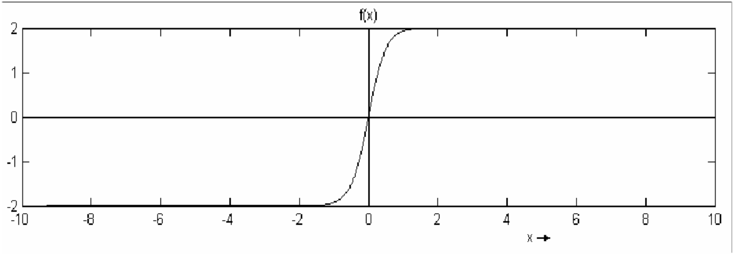
2. Sigmoid function:



$$f(x) = \frac{a}{1 + \exp(-bx)}, \;\; a, b > 0$$

Typical Activation functions

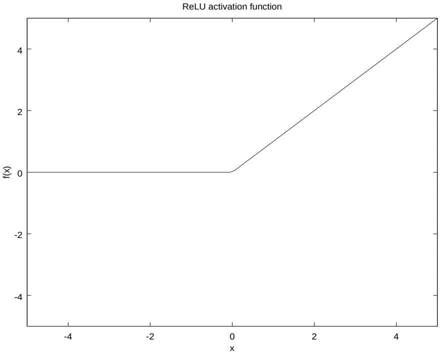
3. tanh



$$f(x) = a\mathsf{tanh}(bx), \;\; a, b > 0$$

Typical Activation functions

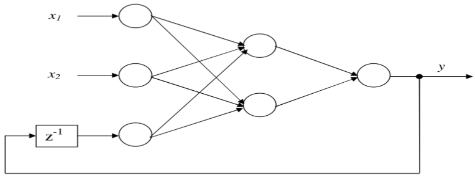
4. ReLU (Rectified Linear Unit)



$$f(x) = \max(0, x)$$

Recurrent networks

- The network we saw earlier has no feedback.
- Here is an example of a network with feedback

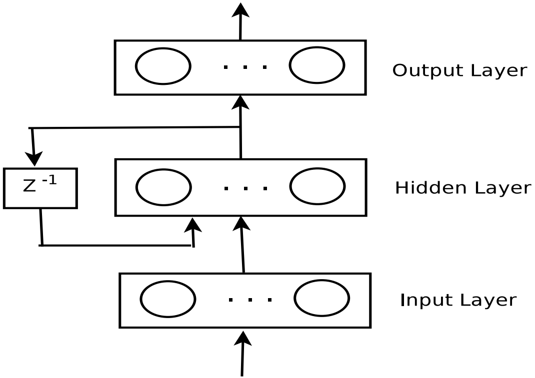


- Can model a dynamical system:

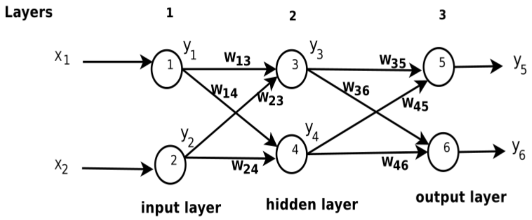
$$y(k) = f(y(k-1), x_1(k), x_2(k))$$

Recurrent Networks

- Here is another example - a general three layer recurrent network



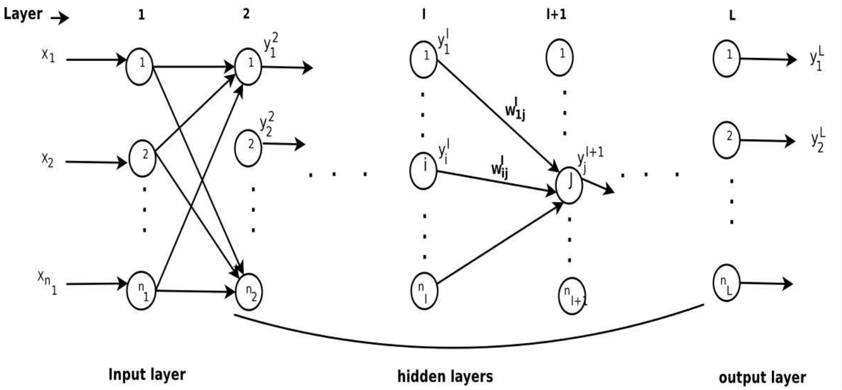
- We start with feedforward networks which can be used as nonlinear pattern classifiers.
- These can always be organized as a layered network.



- This network represents a class of functions from $\Re^2$ to $\Re^2$.

Multilayer feedforward networks

- Here is a general multilayer feedforward network.



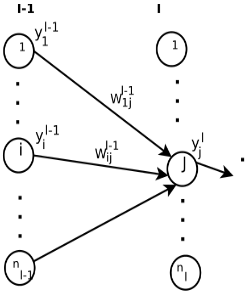
Notation

- L – number of layers
- n_ℓ – number of nodes in layer ℓ, $\ell = 1, \cdots, L$.
- y_i^ℓ – output of i^{th} node in layer ℓ,
  $$i = 1, \cdots, n_\ell,\ \ell = 1, \cdots, L.$$
- w_{ij}^ℓ – weight of connection from node-i, layer-ℓ to node-j, layer-$(\ell + 1)$.
- η_i^ℓ – net input of node-i in layer-ℓ
- Our network represents a function from $\Re^{n_1}$ to $\Re^{n_L}$.
- The input layer gets external inputs.
- The outputs of the output layer are the outputs of the network.

- Let $X = [x_1, \cdots, x_{n_1}]^T$ represent the external inputs to the network.
- The input layer is special and is only to fanout inputs. Hence we take

$$y_i^1 = x_i, \quad i = 1, \cdots, n_1$$

- From layer-2 onwards, units in each layer, successively compute their outputs



- The outputs of a typical unit is computed as

$$
\begin{aligned}
\eta_j^\ell &= \sum_{i=1}^{n_{\ell-1}} w_{ij}^{\ell-1} y_i^{\ell-1} \\
y_j^\ell &= f(\eta_j^\ell)
\end{aligned}
$$

- We can include a 'bias' also for each unit

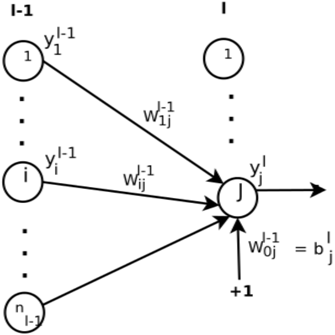
$$\eta_j^\ell \;=\; \sum_{i=1}^{n_{\ell-1}} w_{ij}^{\ell-1} y_i^{\ell-1} \;+\; b_j^\ell$$

- We can think of bias as an extra input and write

$$\eta_j^\ell \;=\; \sum_{i=0}^{n_{\ell-1}} w_{ij}^{\ell-1} y_i^{\ell-1}$$

where, by notation, $w_{0j}^{\ell-1} = b_j^\ell$ and $y_0^\ell = +1$, $\forall\ \ell$.

- This can be shown in the figure as below.



Computing Output of Network (Forward Pass)

- Given inputs, $x_1, \cdots, x_{n_1}$, the outputs are computed as follows.
- For the input layer: $\quad y_i^1 = x_i, \ i = 1, \cdots, n_1$.
- For $\ell = 2, \ \cdots, \ L$, we now compute

$$\begin{aligned} \eta_j^\ell &= \sum_{i=1}^{n_{\ell-1}} w_{ij}^{\ell-1} y_i^{\ell-1} \\ y_j^\ell &= f(\eta_j^\ell) \end{aligned}$$

- The $y_1^L, \ \cdots, y_{n_L}^L$ form the final outputs of the network.

- The network represents functions from $\Re^{n_1}$ to $\Re^{n_L}$.
- The,
  $w_{ij}^{\ell},\ i = 0, \cdots, n_{\ell},\ j = 1, \cdots, n_{\ell+1},\ \ell = 1, \cdots, L-1,$
  are the parameters of the network.
- Let W represent all these parameters.
- The y_i^L are functions of W and the external inputs X, though we may not always explicitly show it in the notation.
- **To get a specific function we need to learn appropriate weights**.

Supervised Learning

- Given a training set $\mathcal{D} = \{(X_i, d_i), i = 1, 2, \cdots\}$, we want to *infer* a 'model' or function f such that on a new data item, X, we *predict* $d = f(X)$. (Note change in notation)

Input/Pattern (X)$\rightarrow$ $\boxed{\text{model (W)}}$ $\rightarrow$ output/label (y)

- This is a discriminative model
- We need to learn the 'optimal' values of parameters, W, using the training data.
- We can use it for classification or regression.

Empirical Risk Minimization

- We employ empirical risk minimization:

$$\min_{W} \; J(W) = \frac{1}{N}\sum_{i=1}^{N} L(y(X_i, W), d_i)$$

- A popular loss function: $L(a, b) = ||(a - b)||^2$

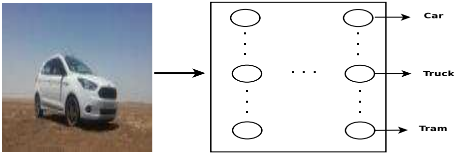
Risk Minimization with Squared Error loss

- Consider network with single output node, sigmoidal activation.
- Let $h(X) = y(X, W)$ be the output.
- Suppose we use it as a 2-class classifier.
- We take d to be binary.
- Then we want W to minimize

$$E\left[(y(X, W) - d)^2\right]$$

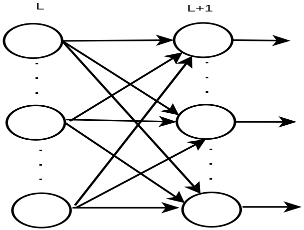
- Then, we know, we are essentially modelling the posterior probability (of class 1).

- Now consider a network for multi-class case.



- We can give d as a vector with one component 1 and others zero.
- The minimizer of squared error loss would again be the (vector-valued) posterior probability function.
- Problem: The network output may not be a probability distribution

Softmax output layer



- We add an extra layer.

$$y_i^{L+1} = \frac{e^{\beta y_i^L}}{\sum_j e^{\beta y_j^L}}$$

- All outputs in the last layer are now positive and sum to one.

Softmax function

- As we saw, the function $f : \Re^m \rightarrow \Re^m$ with

$$f_i(x) = \frac{e^{\beta x_i}}{\sum_j e^{\beta x_j}}$$

  is a smooth approximation to the maximum of a given set of numbers.
- Most neural networks for multi-class classification use softmax layer as the final output layer.
- However, note that this layer would have no weights to be learnt.

Learning a neural network

- Suppose we have training data $\{(X^i, d^i),\ i = 1, \cdots, N\}$, where $X^i = [x_1^i, \cdots, x_m^i]^T \in \Re^m$ and $d^i = [d_1^i, \cdots, d_{m'}^i]^T \in \Re^{m'}$.
- We want to learn a neural network to represent this function.

- We can use a L layer network with $n_1 = m$ and $n_L = m'$.
- L and $n_2, \cdots, n_{L-1}$, are parameters which we fix (for now) arbitrarily.
- We assume that nodes in all layers including output layer use the sigmoid activation function.
- Hence we take $d^i \in [0,\ 1]^{m'},\ \forall i$.
- We can always linearly scale the output as needed.

  (Or we can also use a linear activation function for output nodes).

- Let $y^L = [y_1^L, \cdots, y_{n_L}^L]^T$ be the output.
- We should actually write $y^L(W, X)$, $y_i^L(W, X)$ and so on.
- We want to minimize empirical risk given by

$$
\begin{aligned}
J(W) &= \frac{1}{N} \sum_{i=1}^{N} ||y^L(W, X^i) - d^i||^2 \\
&= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{j=1}^{m'} (y_j^L(W, X^i) - d_j^i)^2 \right)
\end{aligned}
$$

- Same as finding W to minimize

$$J(W) = \sum_{i=1}^{N} J_i(W), \qquad \text{where}$$

$$J_i(W) = \frac{1}{2} \sum_{j=1}^{m'} (y_j^L(W, X^i) - d_j^i)^2$$

- J_i is the square of the error between the output of the network and the desired output for the training example X^i.

- Can use gradient descent to find minimizer of J.
- This gives us the following learning algorithm

$$\begin{aligned} W(t+1) &= W(t) - \lambda \nabla J(W(t)) \\ &= W(t) - \lambda \sum_{i=1}^{N} \nabla J_i(W(t)) \end{aligned}$$

where t is the iteration count and λ is the step-size parameter.

- In terms of the individual weights, the gradient descent is

$$w_{ij}^{\ell}(t+1) = w_{ij}^{\ell}(t) \; - \; \lambda \sum_{s=1}^{N} \frac{\partial J_s}{\partial w_{ij}^{\ell}}(W(t))$$

- We need activation function to be differentiable.
- Gradient descent may give us only a local minimum.

- Our algorithm is

$$w^{\ell}_{ij}(t+1) = w^{\ell}_{ij}(t) \;-\; \lambda \sum_{s=1}^{N} \frac{\partial J_s}{\partial w^{\ell}_{ij}}(W(t))$$

- This is generally called the batch algorithm.
- Often N is very large
- Makes learning slow and computationally expensive.

Stochastic Gradient Descent

- We actually want to minimize

$$J(W) = E\left[||y(X, W) - d||^2\right]$$

- If examples are *iid* then for any random (X^i, d^i), $EJ_i(W) = J(W)$.
- Hence, we may be able to use ∇J_i (for one random i) in place of ∇J.

Stochastic Gradient Descent

- This gives us the following algorithm

$$w_{ij}^{\ell}(t+1) = w_{ij}^{\ell}(t) \; - \; \lambda \, \frac{\partial J_t}{\partial w_{ij}^{\ell}}(W(t))$$

  where $J_t(W) = ||y(X(t), W) - d(t)||^2$
  $(X(t), d(t))$ – the random example at t.
- This is also referred to as the incremental or online version.
- Often we may simply go over all examples in sequence. (One such pass is called an epoch).

MiniBatch Algorithm

- This online stochastic gradient algorithm is using a gradient estimate that may have high variance.
- A good compromise is the minibatch version.

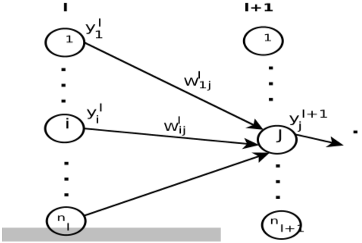
$$w_{ij}^{\ell}(t+1) = w_{ij}^{\ell}(t) \; - \; \lambda \sum_{s=1}^{N_m} \frac{\partial J_s}{\partial w_{ij}^{\ell}}(W(t))$$

  N_m – size of minibatch.
- Here we use the batch algorithm but with only a few examples each time.
- The minibatch size is also a parameter.
- This is also a stochastic gradient descent but with possibly reduced variance.

- All we need now is ∇J_i, for any training sample.
- Because of the structure of the network, there is an efficient way of computing such partial derivatives.
- We look at this computation for any one training sample.
- From now on we omit explicit mention of the specific training example.

- Let $y^L = [y^L_1, \cdots, y^L_{n_L}]^T$ be the output of the network and let $d = [d_1, \cdots, d_{n_L}]^T$ be the desired output.
- Let
$$J = \frac{1}{2} \sum_{j=1}^{n_L} (y^L_j - d_j)^2$$
- We need partial derivative of J with respect to any weight w^ℓ_{ij}.

- Any weight w_{ij}^{ℓ} can affect J only by affecting the final output of the network.
- In a layered network, the weight w_{ij}^{ℓ} can affect the final output only through its affect on $\eta_j^{\ell+1}$.



- Hence, using the chain rule of differentiation, we have

$$\frac{\partial J}{\partial w_{ij}^{\ell}} = \frac{\partial J}{\partial \eta_j^{\ell+1}} \frac{\partial \eta_j^{\ell+1}}{\partial w_{ij}^{\ell}}$$

- Recall that

$$\eta_j^{\ell+1} = \sum_{s=1}^{n_\ell} w_{sj}^{\ell}\, y_s^{\ell} \quad \Rightarrow \quad \frac{\partial \eta_j^{\ell+1}}{\partial w_{ij}^{\ell}} = y_i^{\ell}$$

- Define

$$\delta_j^{\ell} = \frac{\partial J}{\partial \eta_j^{\ell}}, \;\; \forall j, \ell$$

- Now we get

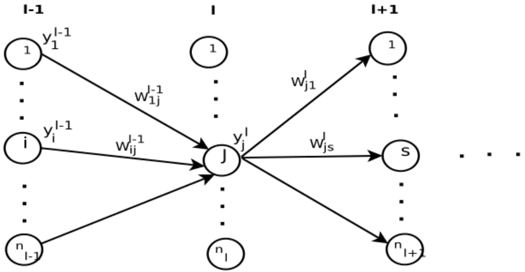
$$\begin{aligned} \frac{\partial J}{\partial w_{ij}^{\ell}} &= \frac{\partial J}{\partial \eta_j^{\ell+1}} \frac{\partial \eta_j^{\ell+1}}{\partial w_{ij}^{\ell}} \\ &= \delta_j^{\ell+1}\, y_i^{\ell} \end{aligned}$$

- We can get all the needed partial derivatives if we calculate δ_j^{ℓ} for all nodes.

- We can compute δ_j^ℓ recursively:

$$\delta_j^\ell = \frac{\partial J}{\partial \eta_j^\ell} \quad = \quad \sum_{s=1}^{n_{\ell+1}} \frac{\partial J}{\partial \eta_s^{\ell+1}} \frac{\partial \eta_s^{\ell+1}}{\partial \eta_j^\ell}$$

$$\delta_j^\ell = \frac{\partial J}{\partial \eta_j^\ell} \quad = \quad \sum_{s=1}^{n_{\ell+1}} \frac{\partial J}{\partial \eta_s^{\ell+1}} \frac{\partial \eta_s^{\ell+1}}{\partial \eta_j^\ell}$$



- We can compute δ_j^ℓ recursively:

$$\delta_j^\ell = \frac{\partial J}{\partial \eta_j^\ell} = \sum_{s=1}^{n_{\ell+1}} \frac{\partial J}{\partial \eta_s^{\ell+1}} \frac{\partial \eta_s^{\ell+1}}{\partial \eta_j^\ell}$$

$$= \sum_{s=1}^{n_{\ell+1}} \frac{\partial J}{\partial \eta_s^{\ell+1}} \frac{\partial \eta_s^{\ell+1}}{\partial y_j^\ell} \frac{\partial y_j^\ell}{\partial \eta_j^\ell}$$

$$= \sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} \, w_{js}^\ell \, f'(\eta_j^\ell)$$

- Recall that the partial derivatives are given by

$$\frac{\partial J}{\partial w_{ij}^{\ell}} = \delta_j^{\ell+1}\, y_i^{\ell}$$

- For the weights, range of ℓ is $\ell = 1, \cdots, (L-1)$.
- Hence we need δ_j^{ℓ} for $\ell = 2, \cdots, L$ and all nodes j.
- Recall the recursive formula for δ_j^{ℓ}

$$\delta_j^{\ell} = \sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1}\, w_{js}^{\ell}\, f'(\eta_j^{\ell})$$

- So, we need to first compute δ_j^{L}.

- By definition,

$$\delta_j^L = \frac{\partial J}{\partial \eta_j^L}$$

- We have

$$J = \frac{1}{2} \sum_{j=1}^{n_L} (y_j^L - d_j)^2$$

- Hence we have

$$\begin{aligned} \delta_j^L = \frac{\partial J}{\partial \eta_j^L} &= \frac{\partial J}{\partial y_j^L} \frac{\partial y_j^L}{\partial \eta_j^L} \\ &= (y_j^L - d_j) f'(\eta_j^L) \end{aligned}$$

- Using the above we can compute $\delta_j^L,\ j = 1,\ \cdots,\ n_L$.
- Then we can compute $\delta_j^\ell,\ j = 1, \cdots, n_\ell$ for $\ell = (L-1),\ \cdots, 2$, recursively, using

$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1}\, w_{js}^\ell \right)\, f'(\eta_j^\ell)$$

- We call δ_j^ℓ the 'error' at node-j layer-ℓ.
- Then we compute all partial derivatives with respect to weights as

$$\frac{\partial J}{\partial w_{ij}^\ell} = \delta_j^{\ell+1}\, y_i^\ell$$

  and hence can update the weights using the gradient descent procedure.

- What we have done so far is calculation of partial derivative of error on any one training sample.
- So, we repeat the procedure for each training example as needed depending on the minibatch size we are using.

- Let us look at the computation of δ_j^ℓ more closely.
- For the output layer, we have

$$\delta_j^L = (y_j^L - d_j) f'(\eta_j^L)$$

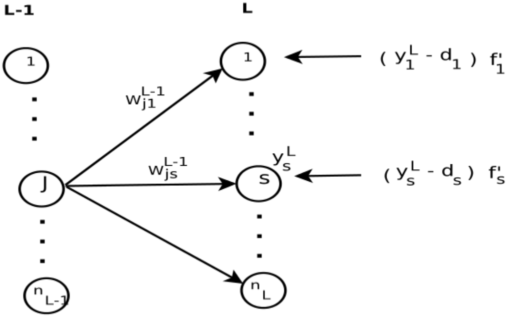
- This is simply the error between the output of the network and the desired output multiplied by f'.
- This is the update equation we had for linear models. (e.g. LMS)

- Consider calculating 'errors' for nodes in layer $L-1$.

$$\delta_j^{L-1} = \left(\sum_{s=1}^{n_L} \delta_s^L \, w_{js}^{L-1} \right) \, f'(\eta_j^{L-1})$$

- We calculate a weighted sum of 'errors' of nodes in Layer-L and multiply by f' of the current node.
- We can look at it graphically as follows.

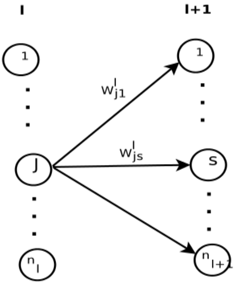
$$\delta_j^{L-1} = \left(\sum_{s=1}^{n_L} \left((y_s^L - d_s) f'(\eta_s^L) \right) \; w_{js}^{L-1} \right) \; f'(\eta_j^{L-1})$$



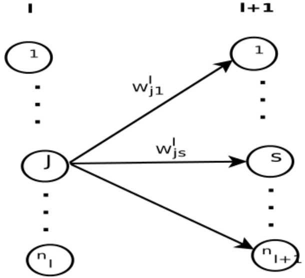
- Now, for every layer backwards (that is, $\ell = L-1, \cdots, 1$), we do similar computation

$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} \, w_{js}^\ell \right) \, f'(\eta_j^\ell)$$

$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} \, w_{js}^\ell \right) \; f'(\eta_j^\ell)$$



$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} \, w_{js}^\ell \right) \, f'(\eta_j^\ell)$$



Backpropagation of errors

Learning the weights

- We can summarize the method as follows.
- We want to learn w_{ij}^{ℓ} to minimize

$$J(W) = \sum_{i=1}^{N'} J_i(W) = \sum_{i=1}^{N'} \frac{1}{2} \left(\sum_{j=1}^{n_L} (y_j^L(X^i, W) - d_j^i)^2 \right)$$

- Using gradient descent for the minimization, we have

$$w_{ij}^{\ell}(t+1) = w_{ij}^{\ell}(t) - \lambda \sum_{s=1}^{N'} \frac{\partial J_s}{\partial w_{ij}^{\ell}}$$

- This process consists of two steps.

Computing output of network (Forward Pass)

- For the input layer: $y_i^1 = x_i^s, \ i = 1, \cdots, n_1$.
- For $\ell = 2, \cdots, L$, we now compute

$$\begin{aligned} \eta_j^\ell &= \sum_{i=1}^{n_{\ell-1}} w_{ij}^{\ell-1} y_i^{\ell-1} \quad \left(\eta^\ell = (W^{\ell-1})^T y^{\ell-1}\right) \\ y_j^\ell &= f(\eta_j^\ell) \quad \left(y^\ell = f(\eta^\ell)\right) \end{aligned}$$

  One matrix-vector product per layer.
- Once we have the output of the network, we then need to compute the 'errors' δ_j^ℓ.

Backpropagation of Errors (Backward Pass)

- At the output layer

$$\delta_j^L = (y_j^L - d_j^s) f'(\eta_j^L)$$

- Now, for layers $\ell = (L-1), \cdots, 2$, we compute

$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} \, w_{js}^\ell \right) \; f'(\eta_j^\ell)$$

- Once all δ_j^ℓ are available, we update the weights by

$$w_{ij}^\ell(t+1) = w_{ij}^\ell(t) \; - \; \lambda \, \delta_j^{\ell+1} \, y_i^\ell$$

- In this process, the squared error loss that we are using affects only the computation of δ_j^L.
- Suppose we use

$$J(W) = L(y^L(X, W), d)$$

Then we get

$$\delta_j^L = \frac{\partial L}{\partial y_j^L} \, f'(\eta_j^L)$$

- The rest of the δ_j^ℓ are all computed the same way as before.
- Thus we can use any other loss function also.
- Though we started with sigmoidal activation, the procedure works with any other (differentiable) activation function

Backpropagation algorithm

- We train the neural network by minimization of empirical risk under squared error loss. (We could use other loss functions)
- We use gradient descent for the minimization.
- Using the network structure, the needed partial derivatives are computed efficiently.
- One forward pass to compute outputs of network and one backward pass to compute all errors and hence all the needed derivatives.
- This algorithm is called **Backpropagation** (or backpropagation of error).

- Suppose we have M_w number of weights and M_l number of nodes in the network.
- Consider the forward computation of obtaining the outputs.
- We essentially do one multiplication per weight and one function evaluation per node.
- The computational complexity is essentially $O(M_w)$ because generally M_w is of the order of M_l^2.
- In the backpropagation of error, the number of computations needed are of the same order as the forward computation.
- Thus we need computational time of the order of M_w to obtain all the partial derivatives.

- Suppose we want to compute the partial derivatives numerically.
- That is, we perturb one w_{ij}^{ℓ} at a time and repeatedly evaluate y^L.
- This would need computing the outputs of the network M_w times.
- Hence we need $O(M_w^2)$ computation to find all partial derivatives.

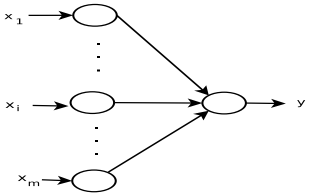
- In deriving the backpropagation algorithm we used

$$\frac{\partial J}{\partial w_{ij}^{\ell}} = \frac{\partial J}{\partial \eta_j^{\ell+1}} \frac{\partial \eta_j^{\ell+1}}{\partial w_{ij}^{\ell}}$$

- We can use this also in obtaining partial derivatives numerically.
- That is, we perturb one η_j^{ℓ} at a time.
- Sometimes called *node perturbation method*.

- This means we may need only $O(M_l)$ forward passes of computing outputs.
- This means we need $O(M_l M_w)$ computation to obtain all partial derivatives.
- Backpropagation is very efficient because it takes only $O(M_w)$ computation.

- Consider a 2-layer network with m input nodes and one output node.

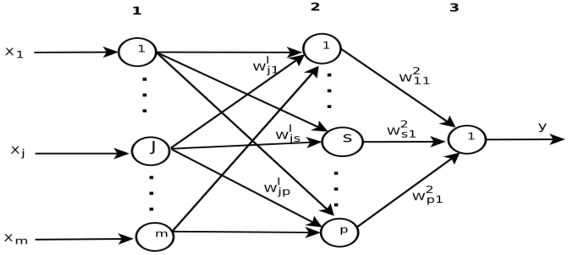


- Such a network represents a 'linear' model

$$y = f\left(\sum_{i=1}^{m} w_i x_i\right)$$

- We can take the inputs to be $\phi_i(X)$ rather than x_i for any fixed functions ϕ_i.

- Consider a 3-layer network with m input nodes, one output node and p nodes in the hidden layer.



- This represents a function from $\Re^m$ to $\Re$.

- The output of this network can be written as

$$y = f\left(\sum_{j=1}^{p} w_{j1}^2 \, f\left(\sum_{i=1}^{m} w_{ij}^1 x_i\right)\right)$$

- Suppose the activation function of output node is linear and all hidden nodes have bias inputs. Then we can rewrite this as

$$y = \sum_{j=1}^{p} \beta_j \, f\left(\sum_{i=1}^{m} w_{ij} \, x_i \, + \, b_j\right)$$

- Now we can ask what kind of functions can be represented like this?
- It can be shown that if p is 'sufficient', then this 3-layer network can well approximate any continuous function over a compact set in $\Re^m$.

Representational abilities

- **Theorem** Let $\phi : \Re \to \Re$ be a bounded, strictly monotonically increasing continuous function. Let $\mathcal{C}(I_m)$ be set of all continuous real valued functions from $I_m = [0,\ 1]^m$ to $\Re$.
  Then, given any $h \in \mathcal{C}(I_m)$ and $\epsilon > 0$, there exists a p and real numbers $\beta_j, b_j, w_{ij},\ i = 1, \cdots, m,\ j = 1, \cdots, p$, such that the function

  $$\hat{h}(X) = \sum_{j=1}^{p} \beta_j \, \phi \left(\sum_{i=1}^{m} w_{ij} x_i \, + \, b_j \right)$$

  satisfies

  $$\sup_{X \in I_m} |h(X) \, - \, \hat{h}(X)| \leq \epsilon$$

- This theorem says that a feedforward network with a single hidden layer (with sufficiently many nodes) can approximate any continuous function to an ϵ-accuracy for any $\epsilon > 0$.
- However, there are no results on how one can estimate the needed p.
- Also, it does not say anything about ease or efficiency of learning this representation.
- But it provides some justification for using these models to represent any function.

- The three layer network represents a function

$$h(X) = \sum_{j=1}^{p} \beta_j \, f\left(\sum_{i=1}^{m} w_{ij} \, x_i \, + \, b_j \right)$$

- This is a linear regression model of the form

$$h(X) = \sum_{j=1}^{p} \beta_j \, \phi_j(X)$$

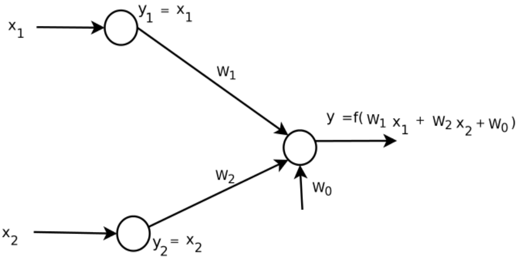
- What is the difference?

Learning Representations

- The main difference is that the basis functions are not fixed beforehand.
- The basis functions themselves are adapted or learnt using the training data.
- We can think of the outputs of the hidden layer as a 'proper' representation of the input so that now we can use a 'linear' model for predicting the target.
- Backpropagatipon algorithm learns proper internal representations.

- Let us look at this using the example of XOR function.
- What is XOR function?
  Given two binary inputs, x_1, x_2, we want the output to be one if and only if exactly one of them is one.
- As you know, a linear classifier or a two layer network can not represent this.
- Let us see this.

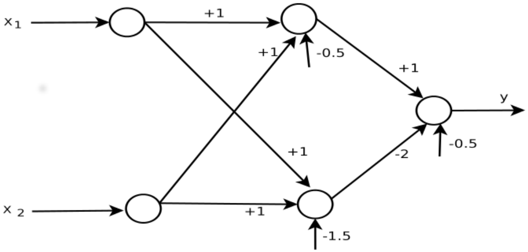
- Take a 2-layer network as below.



- We can not represent XOR with this.

- We can think of XOR as two parts.
- One is to detect when at least one of the two inputs is one.
- Other is to detect when both are one.
- Individually each one is easy.
- It is the combination that is difficult for the network without hidden layer.

- The following 3-layer network represents XOR function.



- The hidden nodes provide a ‘proper’ representation of input.

- By using a neural network, we are adapting the basis rather than choose a fixed basis.
- Is there some advantage?
- One can show that a neural network (whose weights globally minimize the sum of squared errors) would achieve better approximation accuracy than the function learnt using a fixed basis
- That is, the approximation error with a neural network falls faster with N, the number of examples.

- Neural network models are seen to be quite effective for both classification and regression.
- The backpropagation algorithm is quite effective in learning good representations.
- But to learn the appropriate weights, there are many parameters of the network that need to be chosen.
- Also, gradient descent can get stuck in local minima and the initialization could be crucial.
- We next look at a few practical tips to make backpropagation work well.