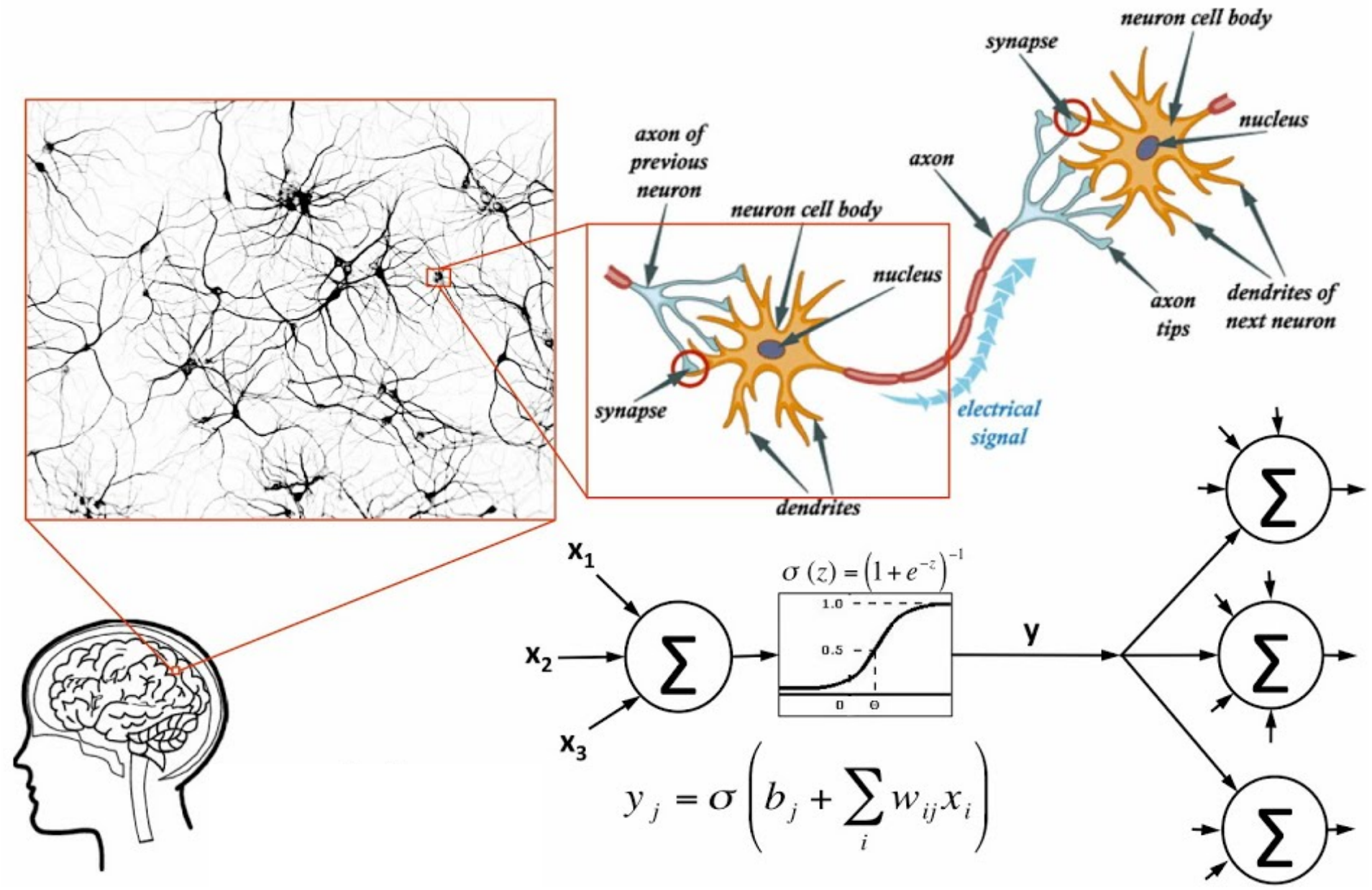
CSI 436/536
Introduction to Machine Learning

Basic Neural Networks

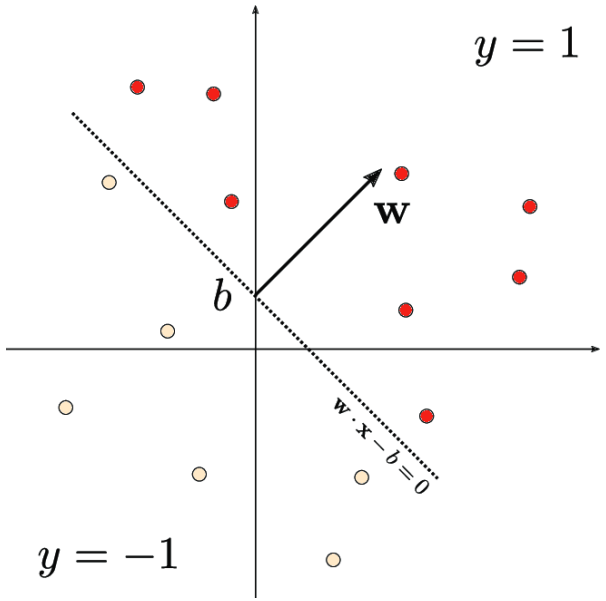
Professor Siwei Lyu
Computer Science
University at Albany, State University of New York

Neural networks

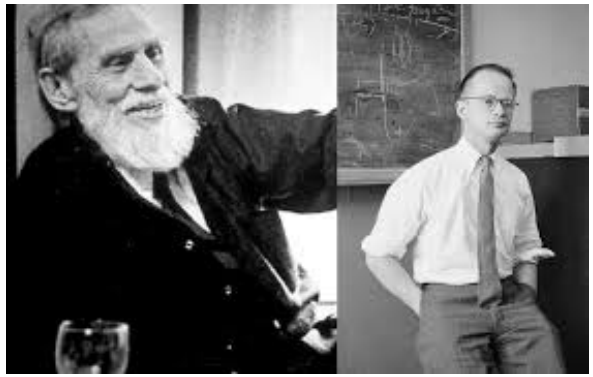


Perceptron

- A single artificial neuron with step-function as activation function is known as ***perceptron***
  - $f(x; w, b) = \text{sign}(w^\top x + b)$
  - Same as linear classification function for LDA, logistic regression, and linear SVM



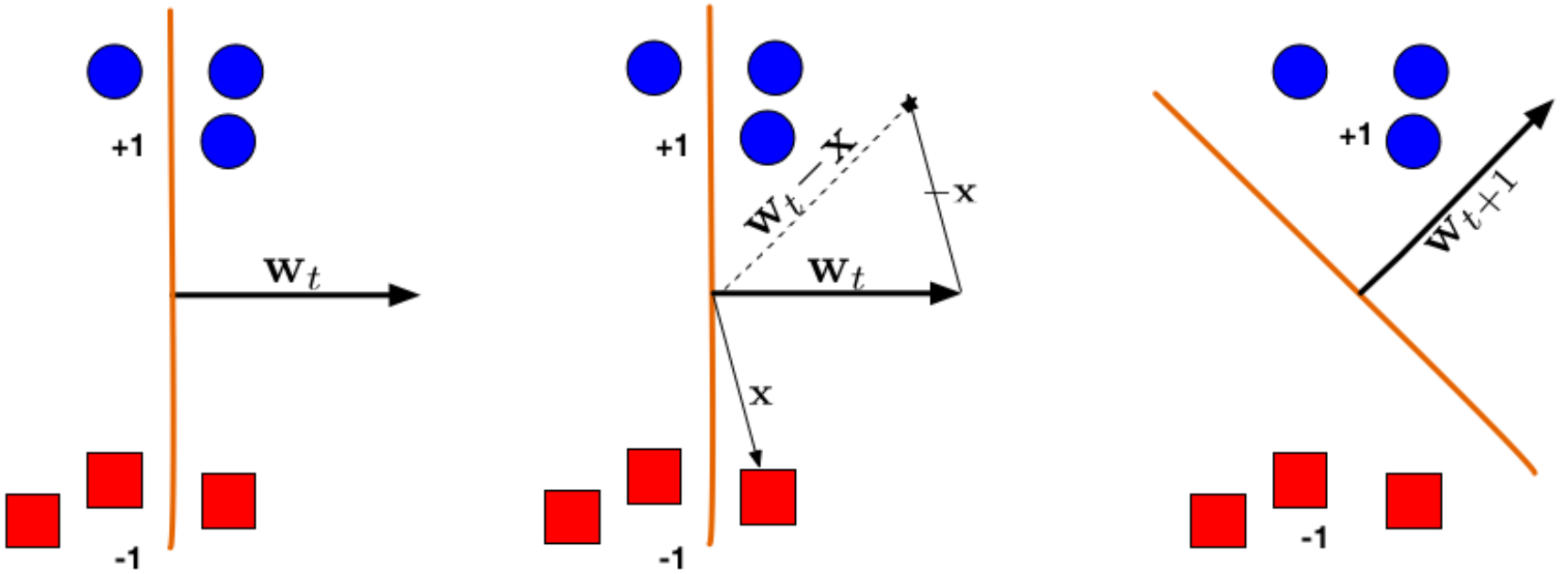
- online training algorithm
  - first developed by McClum & Pitts in the 1950s
  - An instance of online stochastic gradient descent algorithm
  - convergence is theoretically guaranteed



Perceptron algorithm

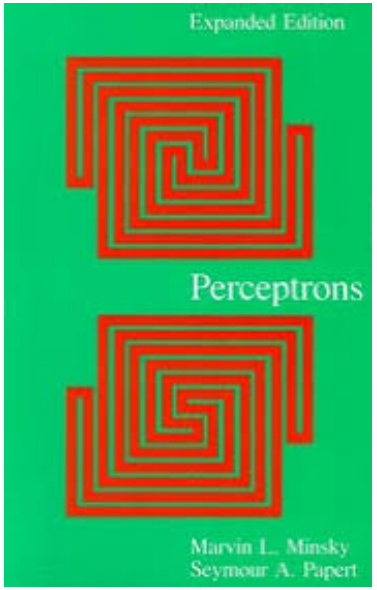
Initialize $\mathbf{w}$ randomly;
while !*convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$;
  if $\mathbf{x} \in P$ *and* $\mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
  end
  if $\mathbf{x} \in N$ *and* $\mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
  end
end

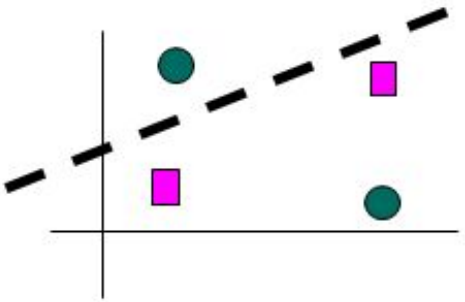
Fix error when occurs

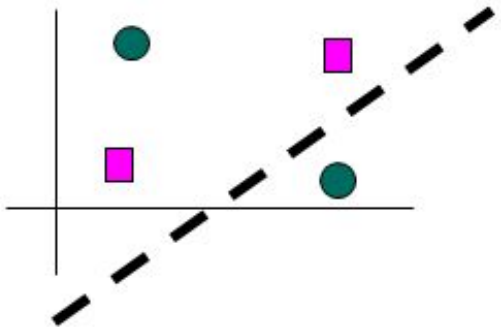


Perceptron algorithm

- A very important algorithm for many reasons
  - Online stochastic gradient algorithm
  - Consistent with "Hebbian learning", fixing error by compensating in the same direction
- Problems
  - (Minsky & Pappert 1962) linear classifier cannot separate XOR type of data (non-separable)



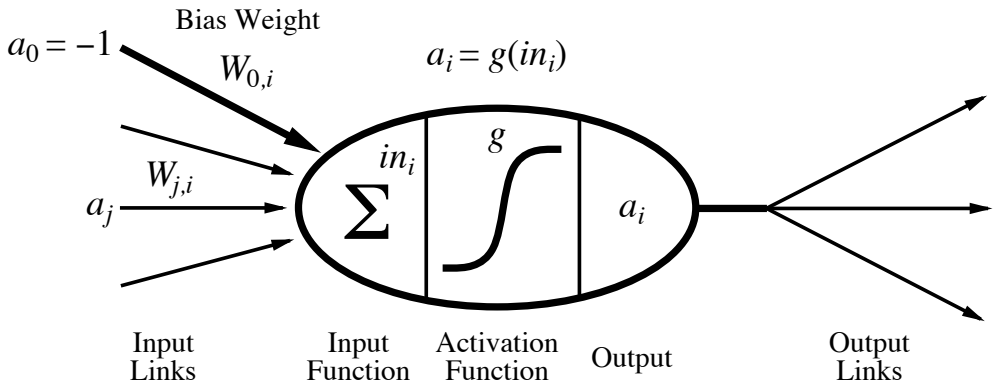


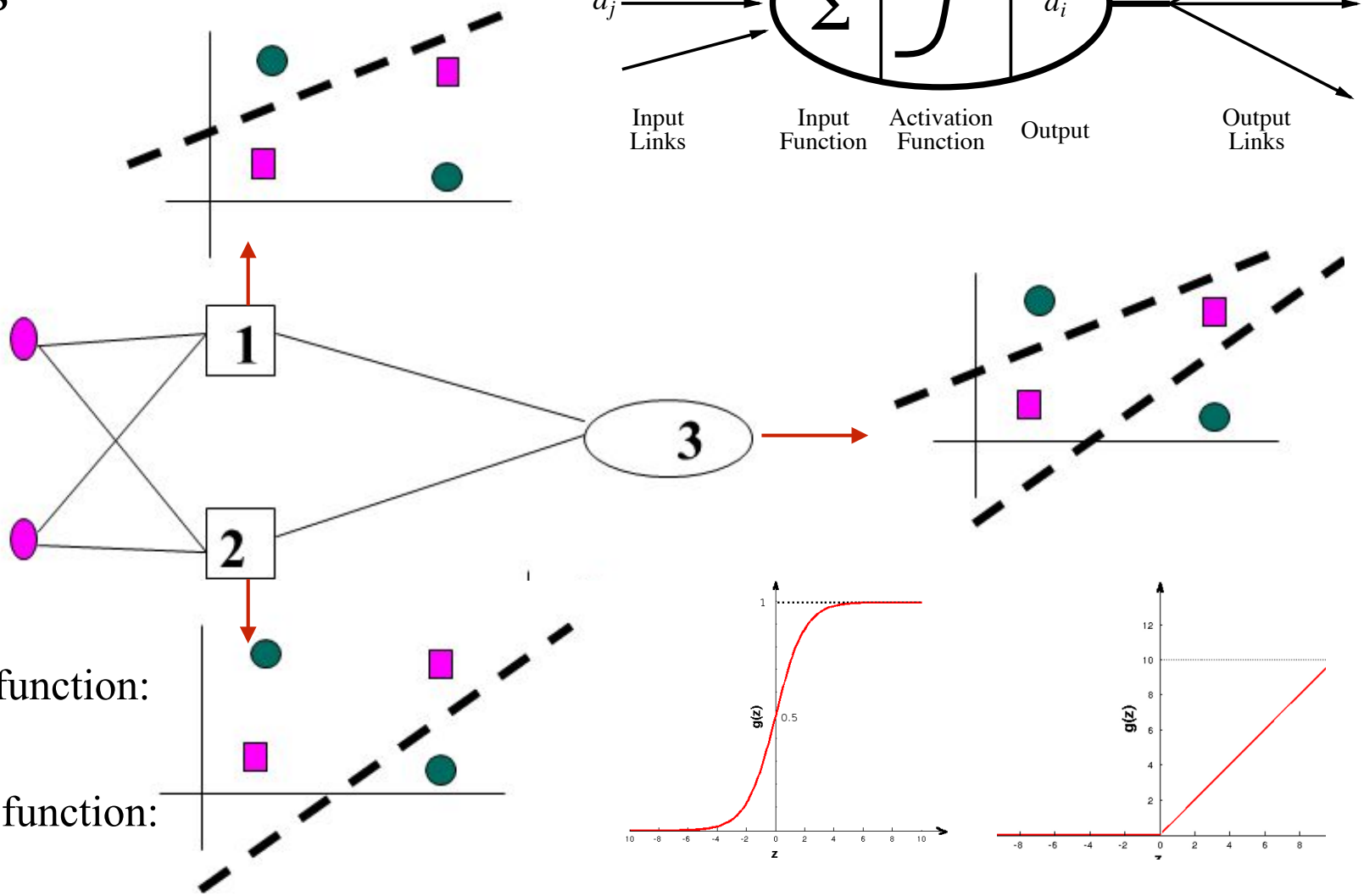


Multi-layer perceptron

- Adding multiple layers
- Introduce smooth activation functions

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$





- Sigmoid function
  $\sigma(x) = (1+e^{-x})^{-1}$
- hyperbolic tangent function:
  $\tanh(x) = 2\sigma(x) - 1$
- rectified linear unit function:
  $\text{relu}(x) = \max(x,0)$

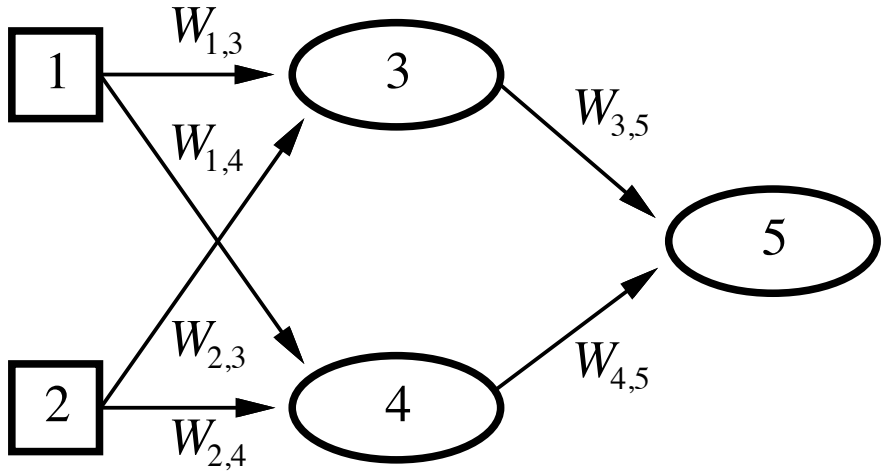
Back propagation

- Training method for multi-layer perceptrons, or feed-forward neural networks
- developed by Rumelhart, Hinto and Williams in 1980s

  - treat NN as a parametric function input to output
  - use training data (input-output pairs) to perform supervise training
  - minimize training error (measured by a loss function) with regards to the NN
  - dynamic programming computation of the gradient to the parameter

feedforward neural network

- feed-forward network = a parameterized family of nonlinear functions
- adjusting weights changes the function: this is how NN is trained

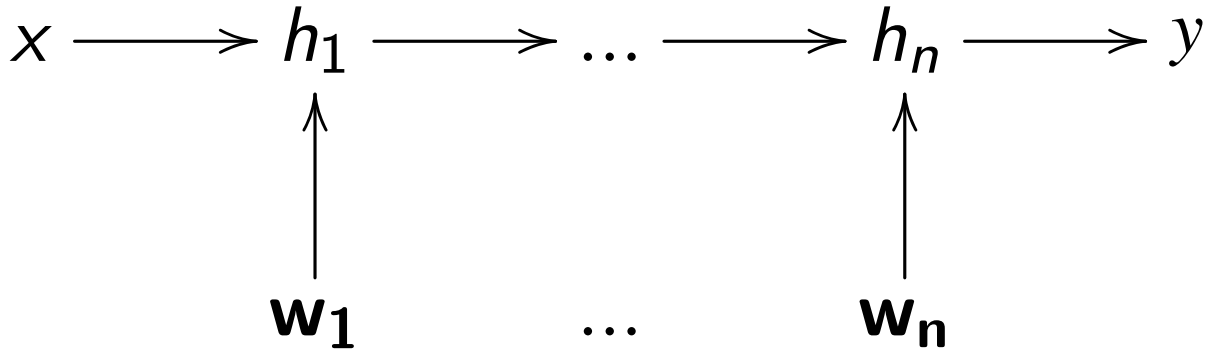


- Computing graph (DAG)
  - input layer
    - input weights
  - hidden layers
    - activation
    - hidden weights
  - output layer

$$
\begin{aligned}
a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\
&= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))
\end{aligned}
$$

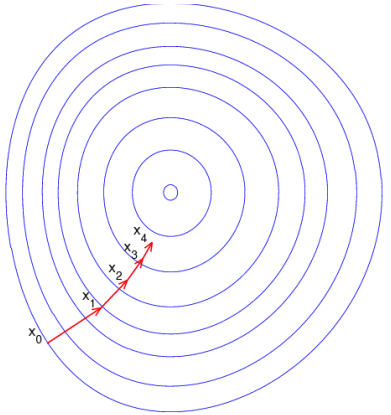
Compute gradient

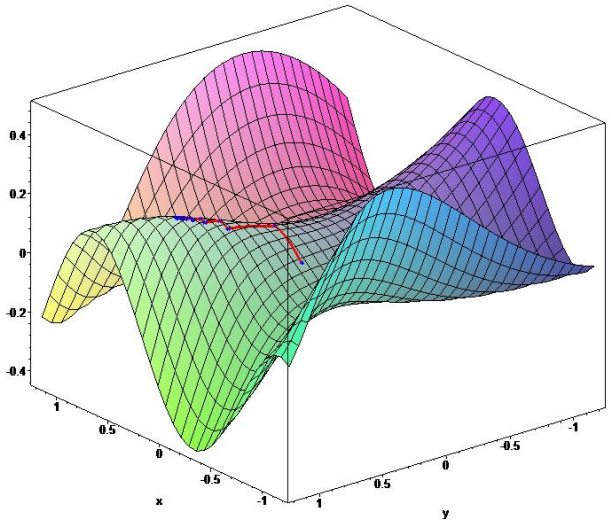
- Network structure x: input data, y: target variable,
  for i=2,…m
  h_i: i-th layer output $h_i(x) = g(w_i^T h_{i-1}(x))$
  $\mathbf{w}_i$: network weight of i-th layer (matrix)
- loss function $L(y, h_n(x))$: L_2 loss, log likelihood, cross-entropy, etc
- Learning objective: with training data
  $\min_{w_1,\cdots,w_n} L(w_1, \cdots, w_n) = \sum_{k=1}^{n} L(y_k, h_n(x_k))$



Optimization by gradient

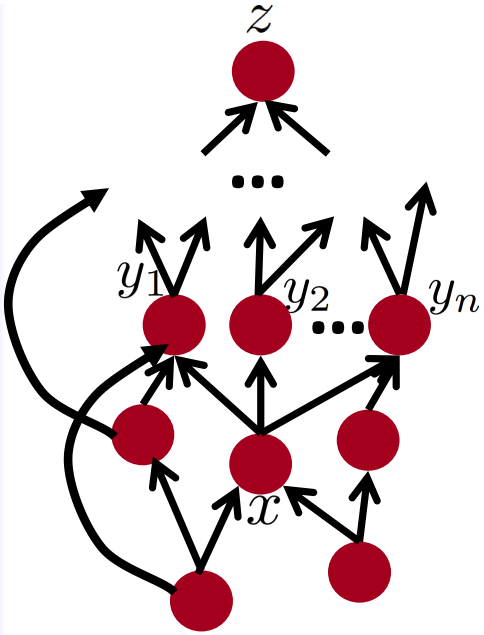
- The learning objective is
$$\min_W L(W) = \sum_{k=1}^{n} L(y_k, h_n(x_k))$$
- We perform stochastic gradient optimization
  - Initializing $W^{(0)}$
  - Iterate until convergence
$$W^{(t)} = W^{(t-1)} - \eta_t \nabla L(W^{(t-1)})$$
  - $\nabla L(W^{(t-1)})$ is the gradient of loss function w.r.t network parameter
  - $\eta_t = \frac{\eta_0}{t+1}$ is the step size
- the key to compute $\nabla L(W^{(t-1)})$ is the ***chain rule*** in calculus

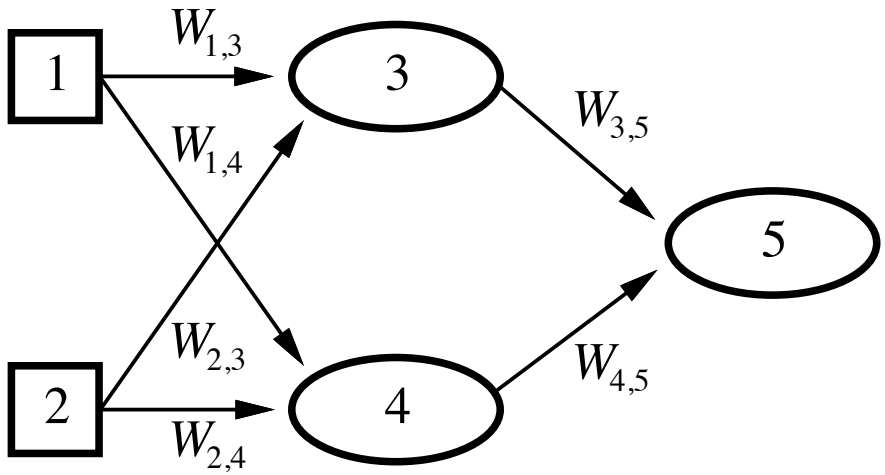




Computation graph

- computation graph: a directed acyclic graph
  - node: variables (inputs and outputs of neurons)
  - edge: dependencies of variables
  - (y_1,…y_n) are children of x



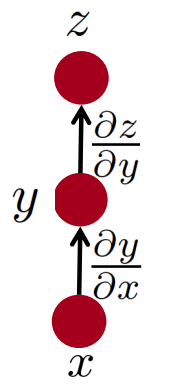


$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

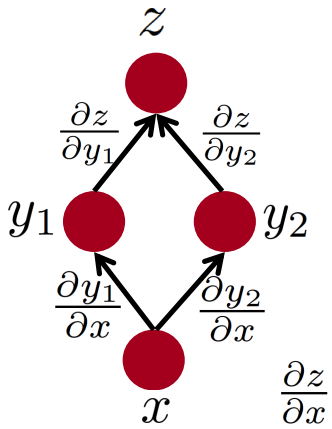
$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

Chain rule

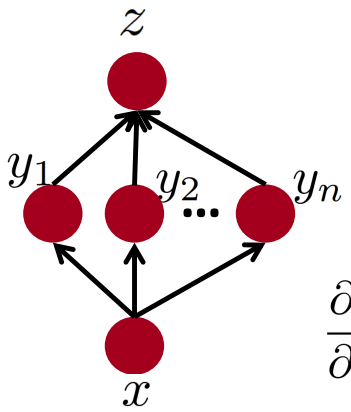
- Chain rule:
  - Multiplication along one route
  - Addition for different branch
- This is a type of algebra rules that can be automated
- Modern ML/NN packages provide automatic gradient derivations
  - Theano, PyTorch, TensorFlow



$$\Delta z = \frac{\partial z}{\partial y}\Delta y$$
$$\Delta y = \frac{\partial y}{\partial x}\Delta x$$
$$\Delta z = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\Delta x$$
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

BP algorithm

- BP = gradient descent update, so we need to compute gradient of weights of each layer
- gradient of loss function w.r.t. w_i using chain rule

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_n}\frac{\partial h_n}{\partial h_{n-1}}\cdots\frac{\partial h_{i+1}}{\partial h_i}\frac{\partial h_i}{\partial w_i}$$

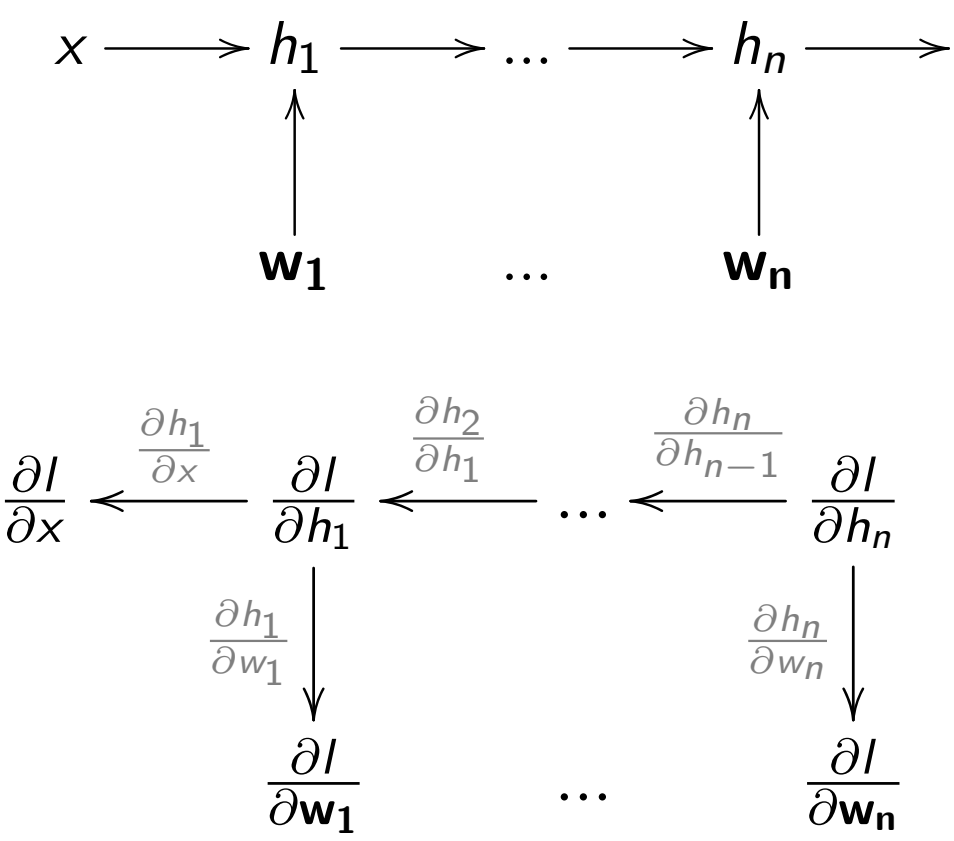
  - recursion

$$\frac{\partial L}{\partial h_{i-1}} = \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial h_{i-1}}$$

    and

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial w_i}$$

  - Dynamic programming to reduce computation



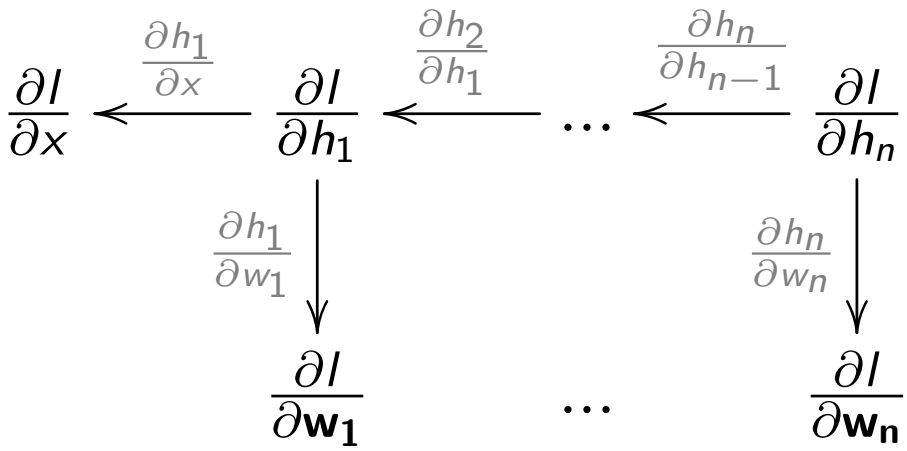
BP algorithm

- BP algorithm compute $\partial L/\partial h_n$ [this shows one step in the iteration over all data and until convergence]
  for i = n:-1:1 (back propagation)

  gradient computation: $\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial w_i}$

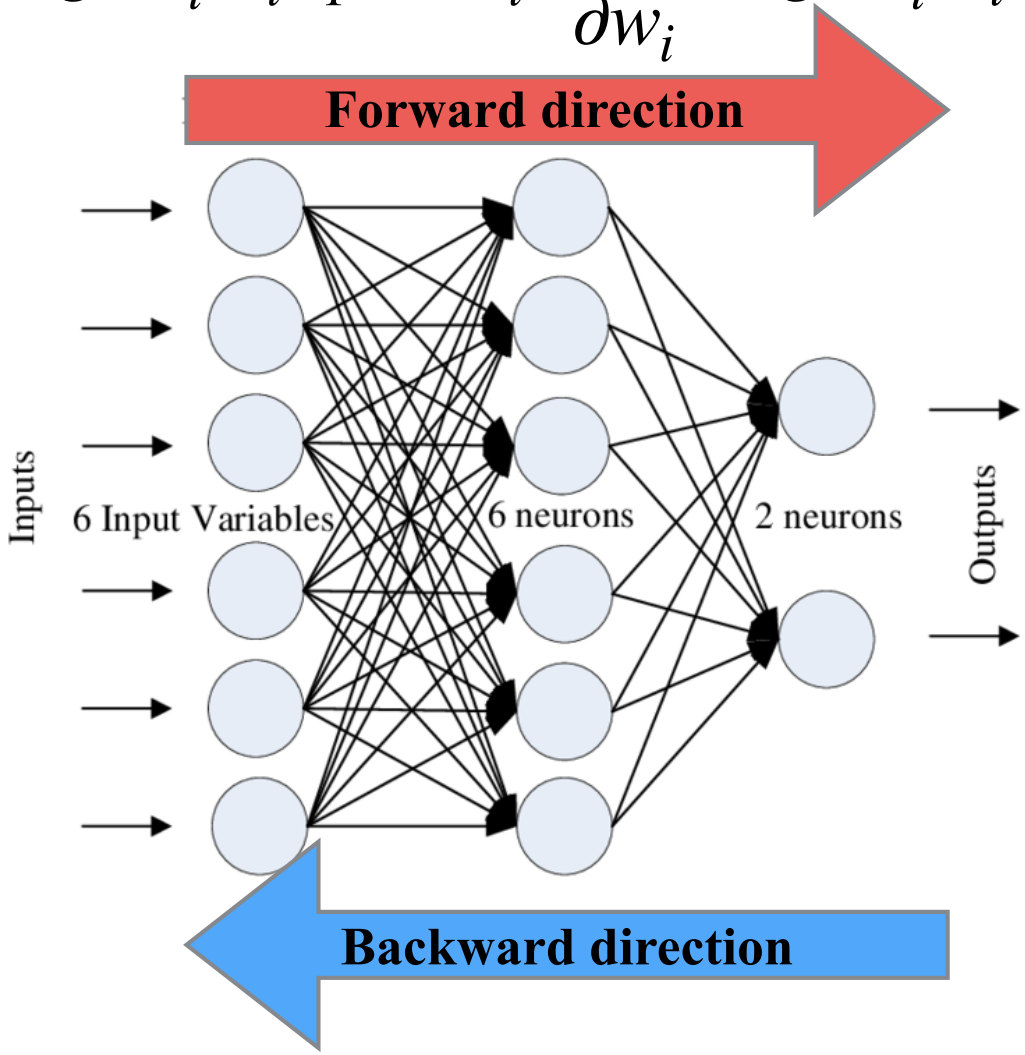
  update current value of w_i with $-\eta_t \frac{\partial L}{\partial w_i}$

  error propagation $\frac{\partial L}{\partial h_{i-1}} = \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial h_{i-1}}$



Compute gradient

- Each layer $h_i(x) = g(w_i^T h_{i-1}(x))$
  - $\frac{\partial h_i}{\partial h_{i-1}} = g'(w_i^T h_{i-1}(x)) w_i$, $\frac{\partial h_i}{\partial w_i} = g'(w_i^T h_{i-1}(x)) h_{i-1}(x)$



drawbacks of BP-trained MLP

- Vanishing gradient

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_n}\frac{\partial h_n}{\partial h_{n-1}}\cdots\frac{\partial h_{i+1}}{\partial h_i}\frac{\partial h_i}{\partial w_i}$$

- The gradient will vanish after several layers of BP
  - Squashing nonlinearity like sigmoid or tanh reduce the range of the values
  - Multiplying smaller values eventually reduce the update to zero (below numerical precision)
- No NN can be effectively trained up to 3 layers — so not very deep model can be used
- This is one reason NN lost favor in ML in late 1990s, which paved the way to SVM

gradient check

- NN code is difficult to debug
- gradient check is a simple trick to make sure no bug in the implementation
  - implement gradient
  - implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon (~10^{-4}) and estimate derivatives

$$g_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2 \times \text{EPSILON}}. \qquad \theta^{(i+)} = \theta + \text{EPSILON} \times \vec{e}_i$$

  - compare the two and make sure they are almost the same

Deriving gradient check

- Taylor expansion

  $f(x+\varepsilon) = f(x) + \varepsilon\nabla f(x) + 0.5\varepsilon^T\nabla^2 f(x)\varepsilon + O(\varepsilon^3)$

  $f(x-\varepsilon) = f(x) - \varepsilon\nabla f(x) + 0.5\varepsilon^T\nabla^2 f(x)\varepsilon + O(\varepsilon^3)$

- So if we use $(f(x+\varepsilon) - f(x))/\varepsilon$ we have second order error, while if we use $(f(x+\varepsilon) - f(x-\varepsilon))/2\varepsilon$ we only have third order error