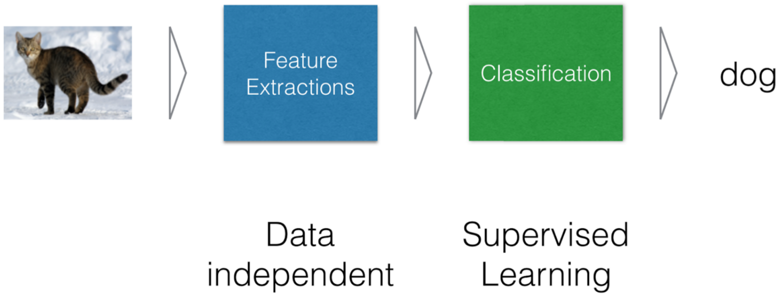
Introduction to Neural Networks

Saeed Reza Kheradpisheh

s_kheradpisheh@sbu.ac.ir

Department of Computer Science
Shahid Beheshti University
Summmer 1398

Typical ML system

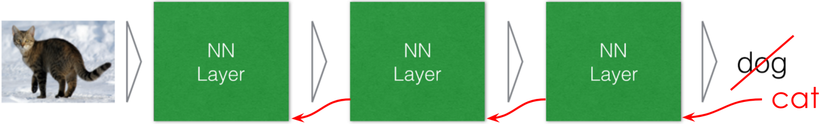


Typical ML system

Data independent

Supervised Learning

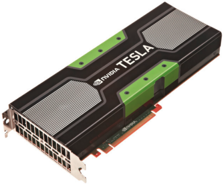
Deep Learning system

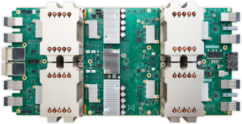


Supervised
Learning

Why Deep Learning Now?

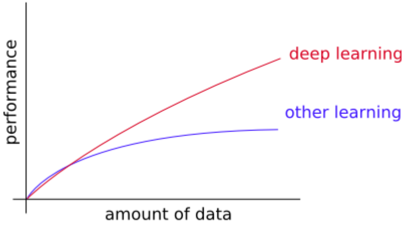
- Computing power (GPUs, TPUs, ...)





Why Deep Learning Now?

- Computing power (GPUs, TPUs, ...)
- Data with labels



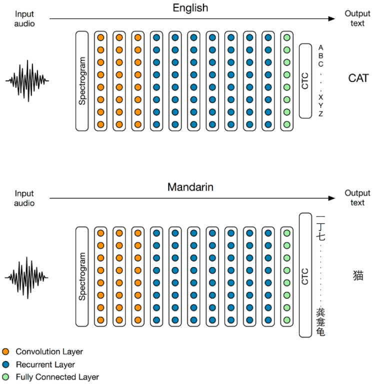
Why Deep Learning Now?

- Computing power (GPUs, TPUs, ...)
- Data with labels
- Open source tools and models

Why Deep Learning Now?

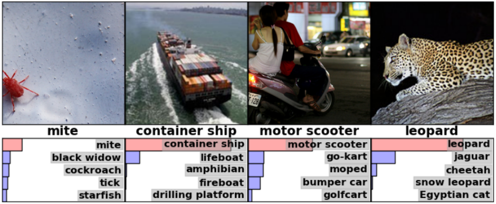
- Computing power (GPUs, TPUs, ...)
- Data with labels
- Open source tools and models
- Better algorithms & understanding

DL Today: Speech-to-Text



[Baidu 2014]

DL Today:Vision



[Krizhevsky 2012]

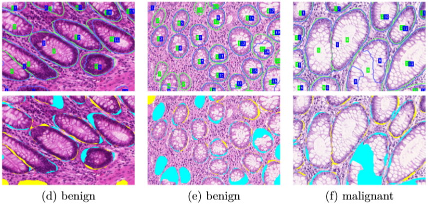
[Ciresan et al. 2013]

[Faster R-CNN - Ren 2015]

[NVIDIA dev blog]

DL Today:Vision

[Stanford 2017]



[Nvidia Dev Blog 2017]

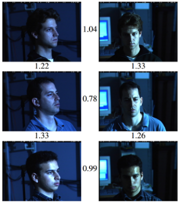
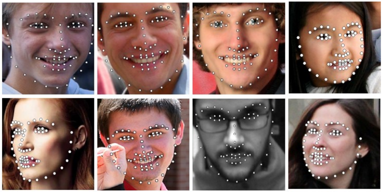


Figure 1. **Illumination and Pose invariance.**

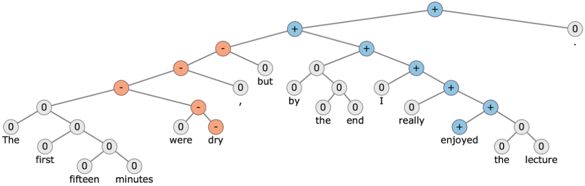
[FaceNet - Google 2015]



[Facial landmark detection CUHK 2014]

DL Today: Natural Language Processing (NLP)

[Google Translate System - 2016]



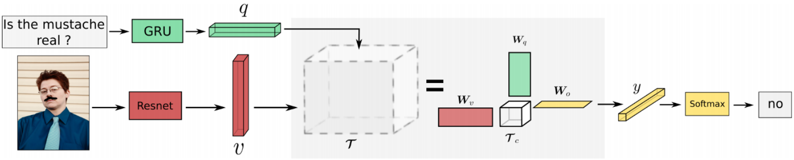
[Socher 2015]

DL Today: Natural Language Processing (NLP)

[Google Inbox Smart Reply]

[Amazon Echo / Alexa]

DL Today: Vision + NLP)



[VQA - Mutan 2017]

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

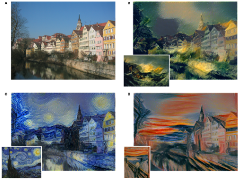
"two young girls are playing with lego toy."

"boy is doing backflip on wakeboard."

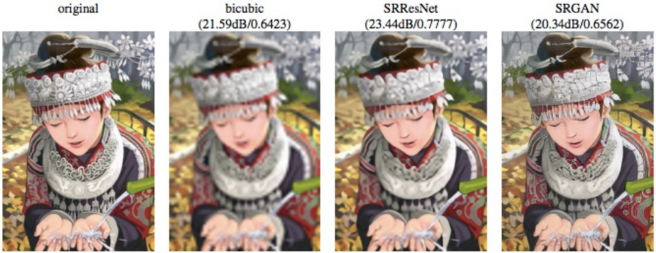
[Karpathy 2015]

DL Today: Image translation

[DeepDream 2015]



[Gatys 2015]



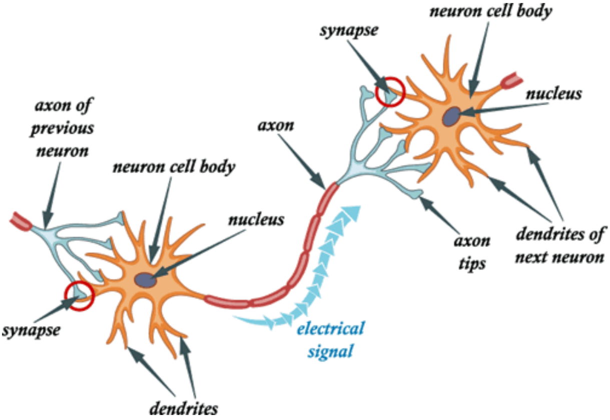
[Ledig 2016]

DL Today: Generative models

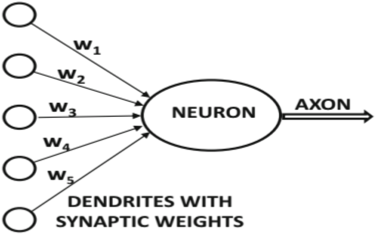
Sampled celebrities [Nvidia 2017]

StackGAN v2 [Zhang 2017]

Biological Neuron

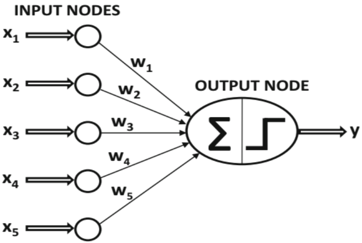


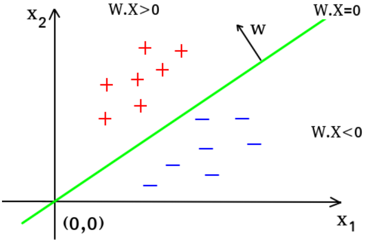
Artificial Neuron



- Neuron output is a function of the inputs.
- Learning occurs by changing the weights to map inputs to outputs.
- Neural networks gain their power by putting together many of basic computing units.

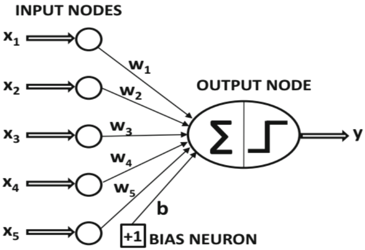
Perceptron

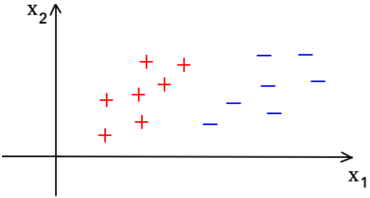




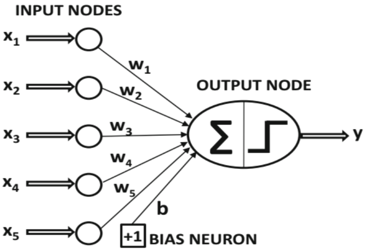
- Input vector: $X = [x_1, ..., x_d]$
- Target output: $Y \in \{-1, +1\}$
- Input weights: $W = [w_1, ..., w_d]$
- Predicted output: $y = sign\{W.X\} = sign\{\sum_{i=1}^{d} w_i x_i\}$

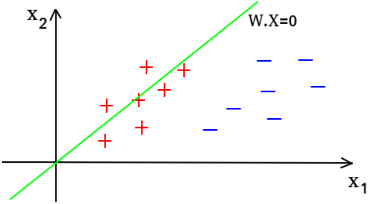
Perceptron with bias



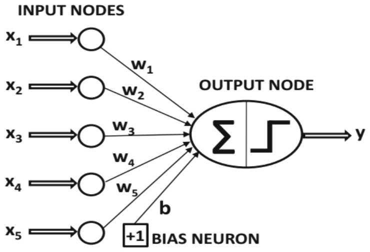


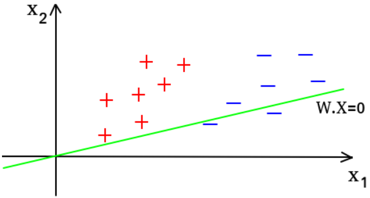
Perceptron with bias



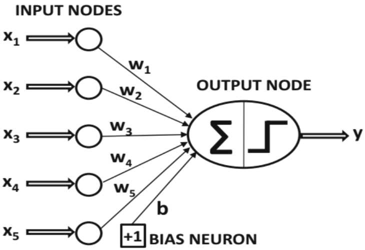


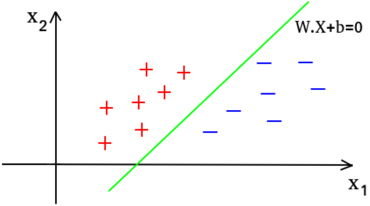
Perceptron with bias





Perceptron with bias





Learning in Perceptrons

Consider a d-dimensional binary classification problem:

- Training set: $D = \{(X_i, Y_i) | i = 1 : N\}$
- Training sample: $X_i = [X_{i1}, ..., X_{id}]$, $Y_i \in \{-1, +1\}$
- Perceptron predicts: $y_i = sign\{W.X_i\}$
- *Loss Function*:

$$L = \sum_{(X_i, Y_i) \in D} (Y_i - y_i)^2 = \sum_{(X_i, Y_i) \in D} (Y_i - sign\{W.X_i\})^2$$

Learning in Perceptrons

Consider a d-dimensional binary classification problem:

- Training set: $D = \{(X_i, Y_i) | i = 1:N\}$
- Training sample: $X_i = [X_{i1}, ..., X_{id}]$, $Y_i \in \{-1, +1\}$
- Perceptron predicts: $y_i = sign\{W.X_i\}$
- *Loss Function*:

$$L = \sum_{(X_i,Y_i) \in D} (Y_i - y_i)^2 = \sum_{(X_i,Y_i) \in D} (Y_i - sign\{W.X_i\})^2$$

- Loss function depends on W and D.

Learning in Perceptrons

Consider a d-dimensional binary classification problem:

- Training set: $D = \{(X_i, Y_i) | i = 1 : N\}$
- Training sample: $X_i = [X_{i1}, ..., X_{id}]$, $Y_i \in \{-1, +1\}$
- Perceptron predicts: $y_i = sign\{W.X_i\}$
- *Loss Function*:

$$L = \sum_{(X_i, Y_i) \in D} (Y_i - y_i)^2 = \sum_{(X_i, Y_i) \in D} (Y_i - sign\{W.X_i\})^2$$

- Loss function depends on W and D.
- As D is given, hence, learning is to find W^* minimizing the loss function:

$$W^* = \underset{W}{\operatorname{argmin}} \sum_{(X_i, Y_i) \in D} (Y_i - sign\{W.X_i\})^2$$

How to find optimum weights?

- At any point $x = [x_1, ..., x_d]$ of a function f, the gradient vector, $\nabla f(x)$, shows the direction of steepest ascend.

How to find optimum weights?

- At any point $x = [x_1, ..., x_d]$ of a function f, the gradient vector, $\nabla f(x)$, shows the direction of steepest ascend.
- For an arbitrary weight vector, $-\nabla_w L$ shows the direction of the steepest descent of the loss function.

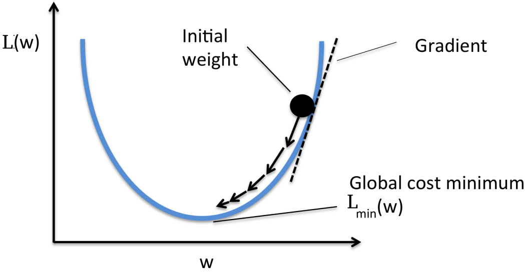
$$\nabla_w L = [\frac{\partial L}{\partial w_1} ..., \frac{\partial f}{\partial w_d}]$$

How to find optimum weights?

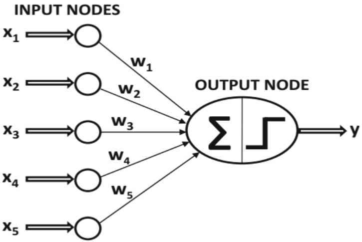
- At any point $x = [x_1, \ldots, x_d]$ of a function f, the gradient vector, $\nabla f(x)$, shows the direction of steepest ascend.
- For an arbitrary weight vector, $-\nabla_w L$ shows the direction of the steepest descent of the loss function.

$$\nabla_w L = [\frac{\partial L}{\partial w_1} \ldots, \frac{\partial f}{\partial w_d}]$$

- Find W^* by starting from a random weight vector and an iterative use of gradient:



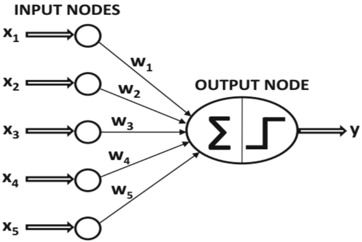
Computing gradients for Perceptron



$$L = \frac{1}{2} \sum_{(X_i, Y_i) \in D} (Y_i - y_i)^2$$

$$= \frac{1}{2} \sum_{(X_i, Y_i) \in D} (Y_i - sign\{W.X_i\})^2$$

Computing gradients for Perceptron

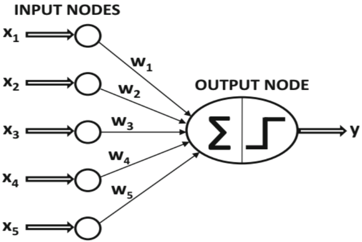


$$\nabla_w L = [\frac{\partial L}{\partial w_1} ..., \frac{\partial f}{\partial w_d}]$$

$$L = \frac{1}{2} \sum_{(X_i, Y_i) \in D} (Y_i - y_i)^2$$

$$= \frac{1}{2} \sum_{(X_i, Y_i) \in D} (Y_i - sign\{W.X_i\})^2$$

Computing gradients for Perceptron



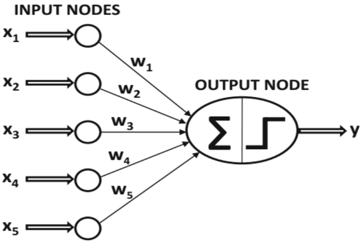
$$\nabla_w L = [\frac{\partial L}{\partial w_1} ..., \frac{\partial f}{\partial w_d}]$$

$$\frac{\partial L}{\partial w_j} = \frac{\partial \frac{1}{2} \sum_{(X_i,Y_i) \in D} (Y_i - y_i)^2}{\partial w_j}$$

$$L = \frac{1}{2} \sum_{(X_i,Y_i) \in D} (Y_i - y_i)^2$$

$$= \frac{1}{2} \sum_{(X_i,Y_i) \in D} (Y_i - sign\{W.X_i\})^2$$

Computing gradients for Perceptron



$$\nabla_w L = [\frac{\partial L}{\partial w_1}, ..., \frac{\partial f}{\partial w_d}]$$

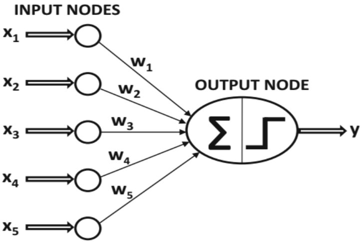
$$\frac{\partial L}{\partial w_j} = \frac{\partial \frac{1}{2} \sum_{(X_i, Y_i) \in D} (Y_i - y_i)^2}{\partial w_j}$$

$$= \frac{1}{2} \sum_{(X_i, Y_i) \in D} \frac{\partial (Y_i - y_i)^2}{\partial w_j}$$

$$L = \frac{1}{2} \sum_{(X_i, Y_i) \in D} (Y_i - y_i)^2$$

$$= \frac{1}{2} \sum_{(X_i, Y_i) \in D} (Y_i - sign\{W.X_i\})^2$$

Computing gradients for Perceptron



$$\nabla_w L = [\frac{\partial L}{\partial w_1} \ldots, \frac{\partial f}{\partial w_d}]$$

$$\frac{\partial L}{\partial w_j} = \frac{\partial \frac{1}{2} \sum_{(X_i,Y_i)\in D} (Y_i - y_i)^2}{\partial w_j}$$

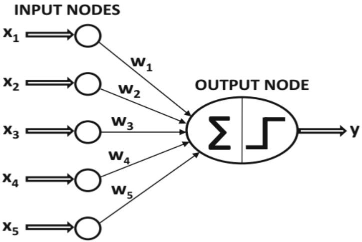
$$= \frac{1}{2} \sum_{(X_i,Y_i)\in D} \frac{\partial (Y_i - y_i)^2}{\partial w_j}$$

$$L = \frac{1}{2} \sum_{(X_i,Y_i)\in D} (Y_i - y_i)^2$$

$$= \frac{1}{2} \sum_{(X_i,Y_i)\in D} \frac{\partial (Y_i - y_i)^2}{\partial y_i} \frac{\partial y_i}{\partial w_j}$$

$$= \frac{1}{2} \sum_{(X_i,Y_i)\in D} (Y_i - sign\{W.X_i\})^2$$

Computing gradients for Perceptron



$$\nabla_w L = [\frac{\partial L}{\partial w_1} ..., \frac{\partial f}{\partial w_d}]$$

$$\frac{\partial L}{\partial w_j} = \frac{\partial \frac{1}{2} \sum_{(X_i,Y_i)\in D} (Y_i - y_i)^2}{\partial w_j}$$

$$= \frac{1}{2} \sum_{(X_i,Y_i)\in D} \frac{\partial (Y_i - y_i)^2}{\partial w_j}$$

$$= \frac{1}{2} \sum_{(X_i,Y_i)\in D} \frac{\partial (Y_i - y_i)^2}{\partial y_i} \frac{\partial y_i}{\partial w_j}$$

$$= - \sum_{(X_i,Y_i)\in D} (Y_i - y_i) \frac{\partial y_i}{\partial w_j}$$

$$L = \frac{1}{2} \sum_{(X_i,Y_i)\in D} (Y_i - y_i)^2$$

$$= \frac{1}{2} \sum_{(X_i,Y_i)\in D} (Y_i - sign\{W.X_i\})^2$$

Computing gradients for Perceptron

The derivative of the Perceptron's predicted output is zero everywhere and is undefined at zero:

$$y_i = sign\{W.X_i\} = sign\{\sum_{j=1}^{d} w_j x_{ij}\}$$

$$\frac{\partial y_i}{\partial w_j} = \frac{\partial sign\{W.X_i\}}{\partial W.X_i} \frac{\partial W.X_i}{\partial w_j} = \frac{\partial sign\{W.X_i\}}{\partial W.X_i} x_{ij}$$

Computing gradients for Perceptron

The derivative of the Perceptron's predicted output is zero everywhere and is undefined at zero:

$$y_i = sign\{W.X_i\} = sign\{\sum_{j=1}^{d} w_j x_{ij}\}$$

$$\frac{\partial y_i}{\partial w_j} = \frac{\partial sign\{W.X_i\}}{\partial W.X_i} \frac{\partial W.X_i}{\partial w_j} = \frac{\partial sign\{W.X_i\}}{\partial W.X_i} x_{ij}$$

Hence, a surrogate gradient is used:

$$\frac{\partial y_i}{\partial w_j} = x_{ij}$$

Thus we have:

$$\frac{\partial L}{\partial w_j} = -\sum_{(X_i,Y_i)\in D} (Y_i - y_i) x_{ij}$$

Gradient Descent Learning for Perceptron

To find optimum weights (W^*):

- Start from a random initial weight vector, W^0.

Gradient Descent Learning for Perceptron

To find optimum weights (W^*):

- Start from a random initial weight vector, W^0.
- Through an iterative manner, use gradients and update the weights:

$$W^{t+1} = W^t + \eta \sum_{(X_i, Y_i) \in D} (Y_i - y_i) X_i$$

Gradient Descent Learning for Perceptron

To find optimum weights (W^*):

- Start from a random initial weight vector, W^0.
- Through an iterative manner, use gradients and update the weights:

$$W^{t+1} = W^t + \eta \sum_{(X_i,Y_i)\in D} (Y_i - y_i)X_i$$

- The laerning rate is controled by η.

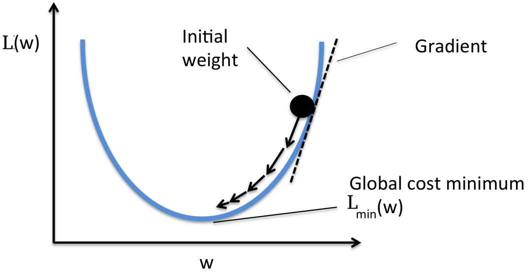
Gradient Descent Learning for Perceptron

To find optimum weights (W^*):

- Start from a random initial weight vector, W^0.
- Through an iterative manner, use gradients and update the weights:

$$W^{t+1} = W^t + \eta \sum_{(X_i, Y_i) \in D} (Y_i - y_i) X_i$$

- The laerning rate is controled by η.



Stocastic Gradient Descent (SGD) for Perceptron

In SGD, learning is performed sample by sample:

1. Shuffle the training set.

Stocastic Gradient Descent (SGD) for Perceptron

In SGD, learning is performed sample by sample:

1. Shuffle the training set.
2. Compute the perceptron's outputy_i for the i-th sample.

Stocastic Gradient Descent (SGD) for Perceptron

In SGD, learning is performed sample by sample:

1. Shuffle the training set.
2. Compute the perceptron's outputy_i for the i-th sample.
3. Update the weights using $W^{t+1} = W^t + \eta(Y_i - y_i)X_i$.

Stocastic Gradient Descent (SGD) for Perceptron

In SGD, learning is performed sample by sample:

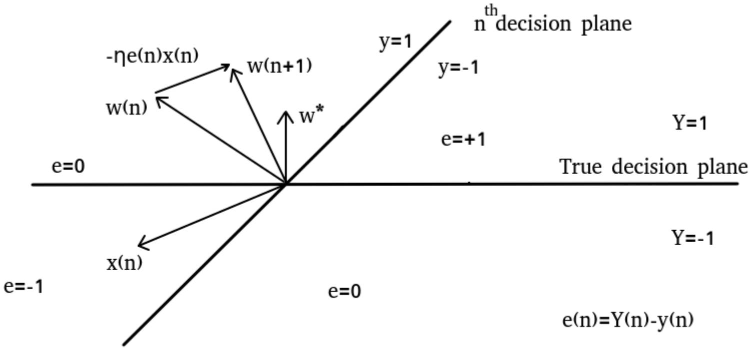
1. Shuffle the training set.
2. Compute the perceptron's outputy_i for the i-th sample.
3. Update the weights using $W^{t+1} = W^t + \eta(Y_i - y_i)X_i$.
4. Repeat steps 2 to 3 for all training samples.

Stocastic Gradient Descent (SGD) for Perceptron

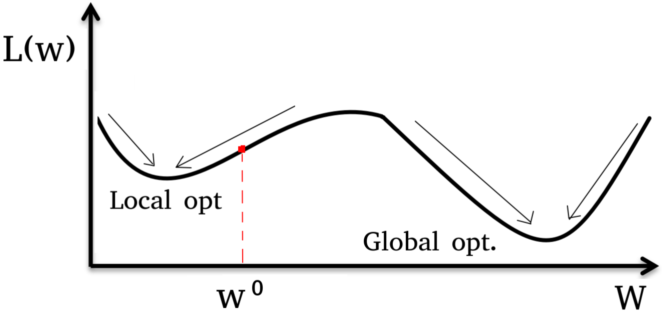
In SGD, learning is performed sample by sample:

1. Shuffle the training set.
2. Compute the perceptron's outputy_i for the i-th sample.
3. Update the weights using $W^{t+1} = W^t + \eta(Y_i - y_i)X_i$.
4. Repeat steps 2 to 3 for all training samples.
5. Jump to step 1 if the totall loss $L = \sum_{(X_i,Y_i)\in D}(Y_i - y_i)^2$ is below a certain value or the maximum number of iteration is reached

SGD

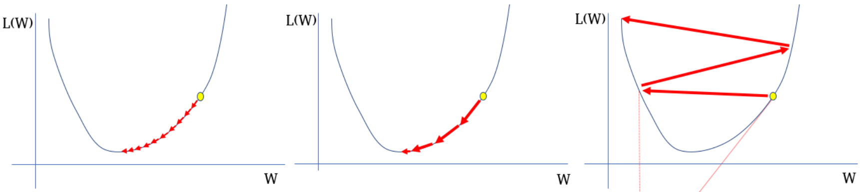

nth decision plane
y=1
y=-1
-ηe(n)x(n)
w(n+1)
w(n)
w*
Y=1
e=+1
e=0
True decision plane
Y=-1
x(n)
e=-1
e=0
e(n)=Y(n)-y(n)

Initial weights matter



Learning rate matters

Too low | **Just right** | **Too high**

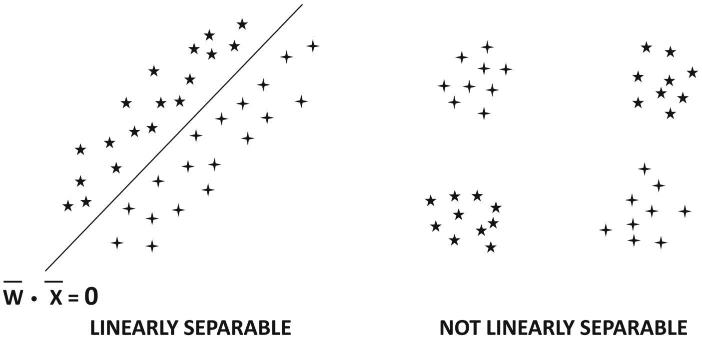


A small learning rate requires many updates before reaching the minimum point

learning rate swiftly reaches the minimum point

Too large of a learning rate causes drastic updates which lead to divergent behaviors

Perceptron solves linearly separable problems

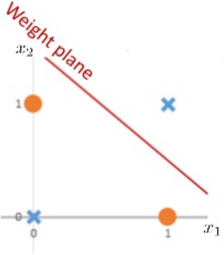


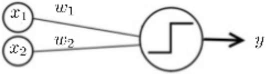
The XOR problem

XOR Problem

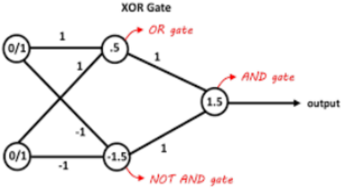
Training Data

x_1	x_2	t
0	0	0
1	0	1
0	1	1
1	1	0



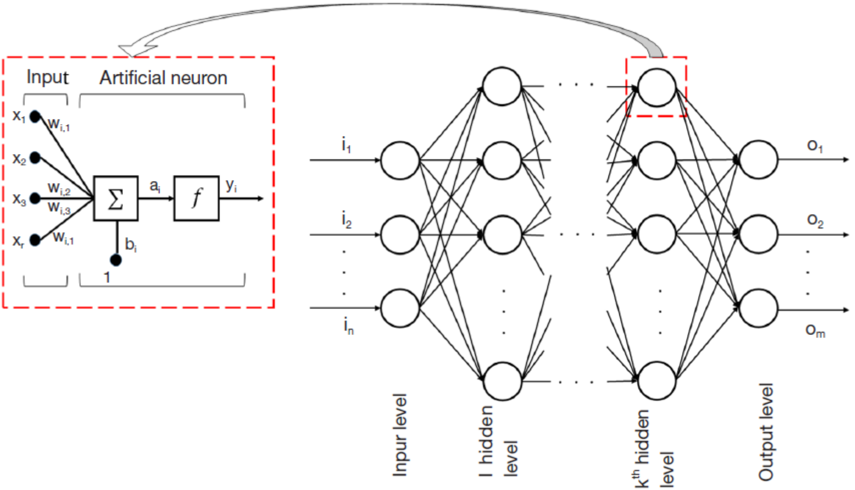


A single perceptron can only solve linear problems.

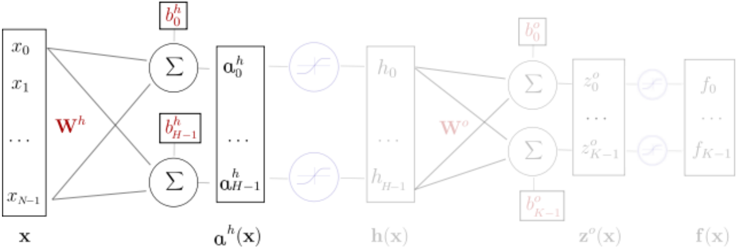


Multi-layer perceptron can solve non-linearly separable problems.

Multi Layer Perceptron (MLP)

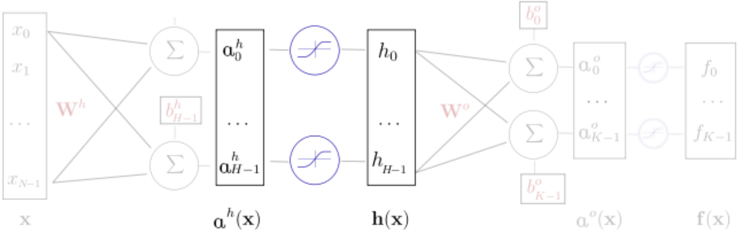


Forward propagation



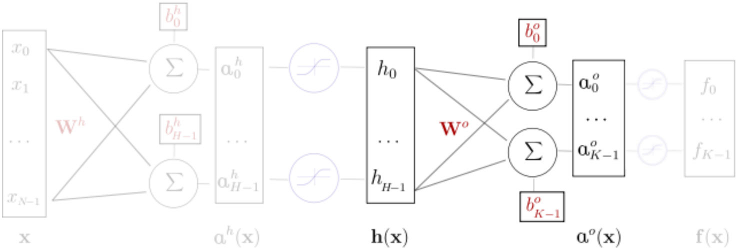
- $a^h(x) = W^h.x + b^h$

Forward propagation



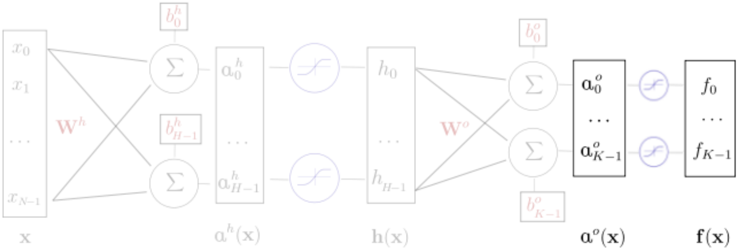
- $a^h(x) = W^h.x + b^h$
- $h(x) = \Phi(a^h(x)) = \Phi(W^h.x + b^h)$

Forward propagation



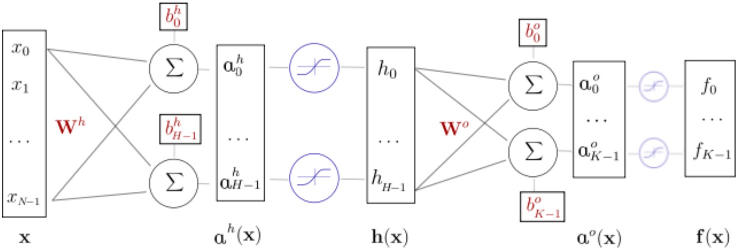
- $a^h(x) = W^h.x + b^h$
- $h(x) = \Phi(a^h(x)) = \Phi(W^h.x + b^h)$
- $a^o(x) = W^o.h(x) + b^o$

Forward propagation



- $a^h(x) = W^h.x + b^h$
- $h(x) = \Phi(a^h(x)) = \Phi(W^h.x + b^h)$
- $a^o(x) = W^o.h(x) + b^o$
- $f(x) = \Phi(a^o(x)) = \Phi(W^o.h(x) + b^o)$

Forward propagation



- $a^h(x) = W^h.x + b^h$
- $h(x) = \Phi(a^h(x)) = \Phi(W^h.x + b^h)$
- $a^o(x) = W^o.h(x) + b^o$
- $f(x) = \Phi(a^o(x)) = \Phi(W^o.h(x) + b^o)$

Multilayer Network as a Computational Graph

- A multilayer network evaluates compositions of functions computed at individual nodes:

$$f(g_1(.), ..., g_k(.)) \tag{1}$$

Multilayer Network as a Computational Graph

- A multilayer network evaluates compositions of functions computed at individual nodes:

$$f(g_1(.), ..., g_k(.)) \tag{1}$$

- The use of nonlinear activation functions is the key to increase the power of multiple layers.

Multilayer Network as a Computational Graph

- A multilayer network evaluates compositions of functions computed at individual nodes:

$$f(g_1(.), \ldots, g_k(.)) \tag{1}$$

- The use of nonlinear activation functions is the key to increase the power of multiple layers.
- A network with a single hidden layer of nonlinear units can compute almost any reasonable function.

Multilayer Network as a Computational Graph

- A multilayer network evaluates compositions of functions computed at individual nodes:

$$f(g_1(.), \ldots, g_k(.)) \tag{1}$$

- The use of nonlinear activation functions is the key to increase the power of multiple layers.
- A network with a single hidden layer of nonlinear units can compute almost any reasonable function.
- The number of hidden units required to do so is rather large, which increases the number of parameters to be learned.

The Power of Function Composition

A multi-layer network that uses only the identity activation function in all its layers reduces to a single-layer network performing linear regression.

$$\overline{h}_1 = \Phi(W_1^T \overline{x}) = W_1^T \overline{x}$$
$$\overline{h}_{p+1} = \Phi(W_{p+1}^T \overline{h}_p) = W_{p+1}^T \overline{h}_p \quad \forall p \in \{1 \ldots k-1\}$$
$$\overline{o} = \Phi(W_{k+1}^T \overline{h}_k) = W_{k+1}^T \overline{h}_k$$

$$\begin{aligned} \overline{o} &= W_{k+1}^T W_k^T \ldots W_1^T \overline{x} \\ &= \underbrace{(W_1 W_2 \ldots W_{k+1})^T}_{W_{xo}^T} \overline{x} \end{aligned}$$

Element-wise Activation Functions

Sign function: $\Phi(a) = sign(a)$

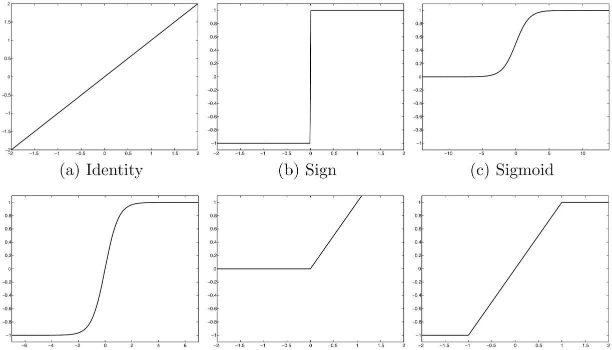
Sigmoid function: $\Phi(a) = \frac{1}{1+e^{-a}}$

Tangh function: $\Phi(a) = \frac{e^{2a}-1}{e^{2a}+1}$

ReLU: $\Phi(a) = max\{a, 0\}$

Hard Tangh:

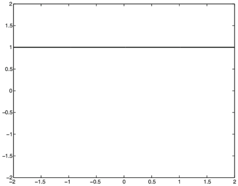
$\Phi(a) = max\{min[v, 1], -1\}$



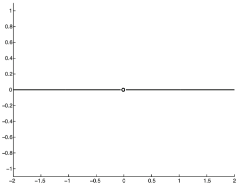
(a) Identity (b) Sign (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

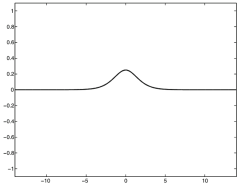
Dervation of Activation Functions



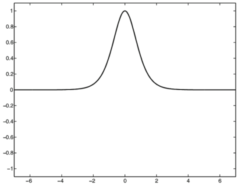
(a) Identity



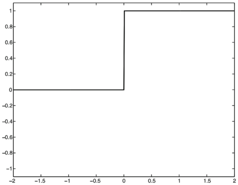
(b) Sign



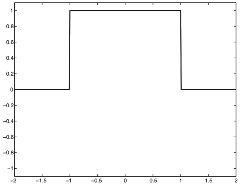
(c) Sigmoid



(d) Tanh



(e) ReLU



(f) Hard Tanh

Dervation of Activation Functions

Assume $o = \Phi(v)$ thus we have:

Sigmoid function:

$$\frac{\partial o}{\partial v} = \frac{\exp(-v)}{(1+\exp(-v))^2}$$

$$\frac{\partial o}{\partial v} = o(1-o)$$

Tangh function:

$$\frac{\partial o}{\partial v} = \frac{4 \cdot \exp(2v)}{(\exp(2v)+1)^2}$$

$$\frac{\partial o}{\partial v} = 1 - o^2$$

Groupe Activation: Softmax function

$$softmax(x) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

Groupe Activation: Softmax function

$$softmax(x) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

- Softmax activation for each neuron is in range [0,1] .

Groupe Activation: Softmax function

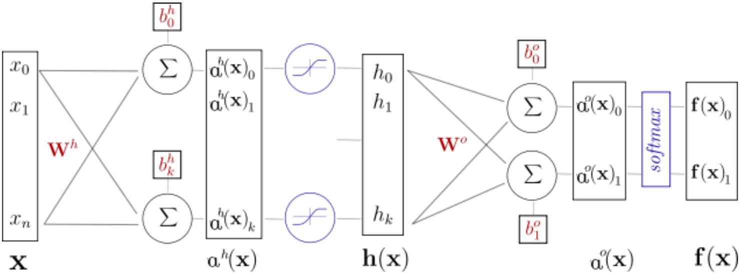
$$softmax(x) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

- Softmax activation for each neuron is in range [0,1] .
- The summation of neurons' activation is 1.

Groupe Activation: Softmax function

$$softmax(x) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

- Softmax activation for each neuron is in range [0,1] .
- The summation of neurons' activation is 1.
- It is ususally used in the output layer.



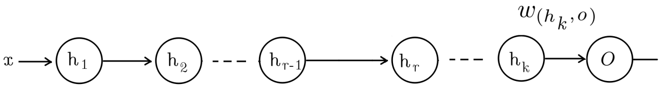
Dervation of the Softmax Function

Assume $o_i = softmax(v_i)$ thus we have:

$$\frac{\partial o_i}{\partial v_j} = \begin{cases} o_i \cdot (1 - o_i) & i = j \\ -o_i \cdot o_j & i \neq j \end{cases}$$

Error Backpropagation

- Consider a sequence of hidden units followed by an output unit.
- To update any weight of the output layer, we use the gradient of the loss function with respect to that weight.
- Consider the Loss Function to be $L(X) = \frac{1}{2}(Y - o)^2$



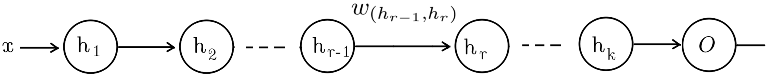
$$\frac{\partial L}{\partial w_{(h_k,o)}} = \frac{\partial L}{\partial o} \cdot \frac{\partial o}{\partial w_{(h_k,o)}} = \Delta(o, o) \cdot h_k \cdot \Phi'(a_o)$$

$$\Delta(o, o) = \frac{\partial L}{\partial o} = -(Y\text{-}O)$$

$$\frac{\partial o}{\partial w_{(h_k,o)}} = \frac{\partial o}{\partial a_o} \frac{\partial a_o}{\partial w_{(h_k,o)}} = h_k \cdot \Phi'(a_o)$$

Error Backpropagation

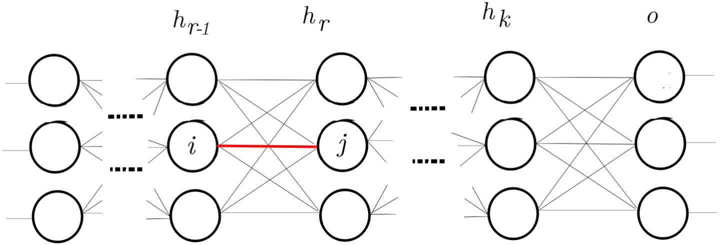
- Consider a sequence of hidden units followed by an output unit.
- To update a connection weight, we should compute the gradient of the loss function with respect to that weight.



$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial L}{\partial o} \cdot \left[\frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$$

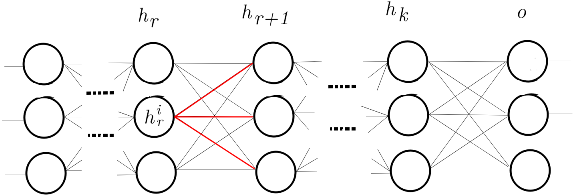
Error Backpropagation

- Now consider a sequence of hidden layers followed by an output unit (P is the set of paths from $h_r$to o):

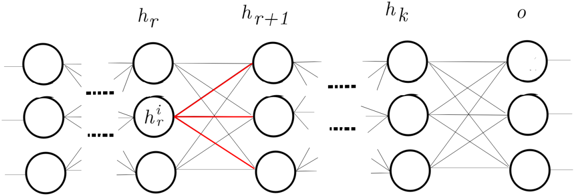


$$\frac{\partial L}{\partial w_{(h^i_{r-1}, h^j_r)}} = \underbrace{\left[\sum_{[h^j_r, h_{r+1}, \ldots h_k, o] \in \mathcal{P}} \frac{\partial L}{\partial o} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\Delta(h^j_r, o) = \frac{\partial L}{\partial h^j_r}} \frac{\partial h^j_r}{\partial w_{(h^i_{r-1}, h^j_r)}}$$

Error Backpropagation

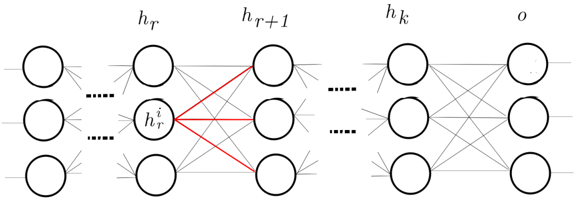


Error Backpropagation



$$\Delta(o, o) = \frac{\partial L}{\partial o}$$

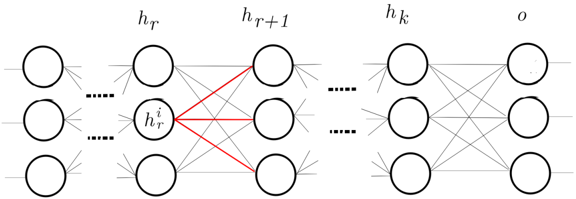
Error Backpropagation



$$\Delta(o, o) = \frac{\partial L}{\partial o}$$

$$\Delta(h_r^i, o) = \frac{\partial L}{\partial h_r^i} = \sum_{h: h_r^i \Rightarrow h_{r+1}} \frac{\partial L}{\partial h} \frac{\partial h}{\partial h_r^i} = \sum_{h: h_r^i \Rightarrow h_{r+1}} \frac{\partial h}{\partial h_r^i} \Delta(h, o)$$

Error Backpropagation



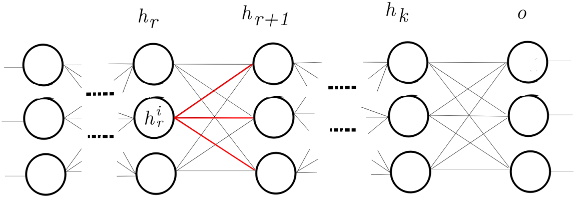
$$a_h = W_h . h_r = \sum_j w_{(h_r^j, h)} h_r^j$$

$$h = \Phi(a_h) = \Phi(\sum_j w_{(h_r^j, h)} h_r^j)$$

$$\frac{\partial h}{\partial h_r^i} = \frac{\partial h}{\partial a_h} \cdot \frac{\partial a_h}{\partial h_r^i} = \frac{\partial \Phi(a_h)}{\partial a_h} \cdot w_{(h_r^i, h)} = \Phi'(a_h) \cdot w_{(h_r^i, h)}$$

$$\frac{\partial h}{\partial w_{(h_r^i, h)}} = \frac{\partial h}{\partial a_h} \cdot \frac{\partial a_h}{\partial w_{(h_r^i, h)}} = \Phi'(a_h) \cdot h_r^i$$

Error Backpropagation

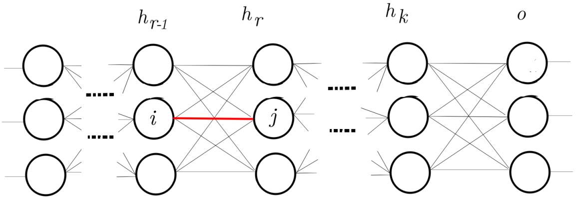


$$\Delta(o,o) = \frac{\partial L}{\partial o}$$

$$\Delta(h_r^i, o) = \frac{\partial L}{\partial h_r^i} = \sum_{h:h_r^i \Rightarrow h_{r+1}} \frac{\partial L}{\partial h}\frac{\partial h}{\partial h_r^i} = \sum_{h:h_r^i \Rightarrow h_{r+1}} \frac{\partial h}{\partial h_r^i}\Delta(h,o)$$

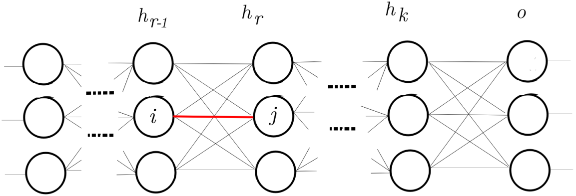
$$\Delta(h_r^i, o) = \sum_{h:h_r^i \Rightarrow h} \Phi'(a_h) \cdot w_{(h_r^i, h)} \cdot \Delta(h,o)$$

Error Backpropagation



$$\frac{\partial L}{\partial w_{(h^i_{r-1}, h^j_r)}} = \underbrace{\left[\sum_{[h^j_r, h_{r+1}, \ldots h_k, o] \in \mathcal{P}} \frac{\partial L}{\partial o} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\Delta(h^j_r, o) = \frac{\partial L}{\partial h^j_r}} \frac{\partial h^j_r}{\partial w_{(h^i_{r-1}, h^j_r)}}$$

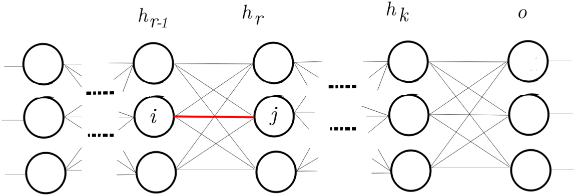
Error Backpropagation



$$\frac{\partial L}{\partial w_{(h^i_{r-1},h^j_r)}} = \underbrace{\left[\sum_{[h^j_r,h_{r+1},\ldots h_k,o]\in\mathcal{P}} \frac{\partial L}{\partial o}\frac{\partial o}{\partial h_k}\prod_{i=r}^{k-1}\frac{\partial h_{i+1}}{\partial h_i}\right]}_{\Delta(h^j_r,o)=\frac{\partial L}{\partial h^j_r}} \frac{\partial h^j_r}{\partial w_{(h^i_{r-1},h^j_r)}}$$

$$\frac{\partial h^j_r}{\partial w_{(h^i_{r-1},h^j_r)}} = h^j_{r-1}\cdot\Phi'(a_{h^j_r})$$

Error Backpropagation



$$\frac{\partial L}{\partial w_{(h_{r-1}^i, h_r^j)}} = \underbrace{\left[\sum_{[h_r^j, h_{r+1}, \ldots h_k, o] \in \mathcal{P}} \frac{\partial L}{\partial o} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\Delta(h_r^j, o) = \frac{\partial L}{\partial h_r^j}} \frac{\partial h_r^j}{\partial w_{(h_{r-1}^i, h_r^j)}}$$

$$\frac{\partial h_r^j}{\partial w_{(h_{r-1}^i, h_r^j)}} = h_{r-1}^j \cdot \Phi'(a_{h_r^j})$$

$$\frac{\partial L}{\partial w_{(h_{r-1}^i, h_r^j)}} = \Delta(h_r^i, o) \cdot h_{r-1}^i \cdot \Phi'(a_{h_r^i})$$

SGD using Backpropagation

For each training sample:

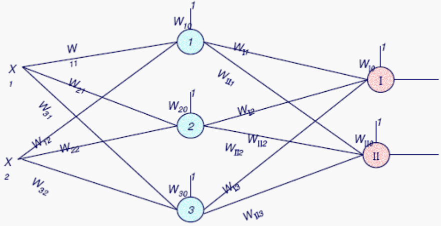
- Compute the forward path.
- Compute $\Delta(o,o)$ for each output neuron.
- Update each connecting weight of the output layer as

$$w_{(h_k,o)} = w_{(h_k,o)} + \eta \cdot \Delta(o,o) \cdot h_k \cdot \Phi'(a_o)$$

- For $r = k, k-1, ..., 1$:
  - Compute $\Delta(h_r^i, o)$ for the i-th neuron at the r-th hidden layer.
  - Update each connecting weight of the i-th neuron at the r-th hidden layer as:

$$w_{(h_{r-1},h_r)} = w_{(h_{r-1},h_r)} + \eta \cdot \Delta(h_r, o) \cdot h_{r-1} \cdot \Phi'(a_{h_r})$$

MLP example



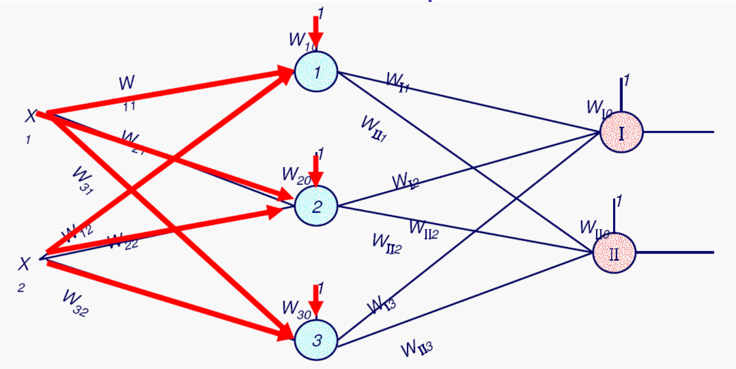
Training Data

Inputs	Target Concept
0,0	0,0
0,1	0,1
1,0	0,1
1,1	1,0

Activation Function

$$f(y) = \frac{1}{1+e^{-y}}$$

MLP example

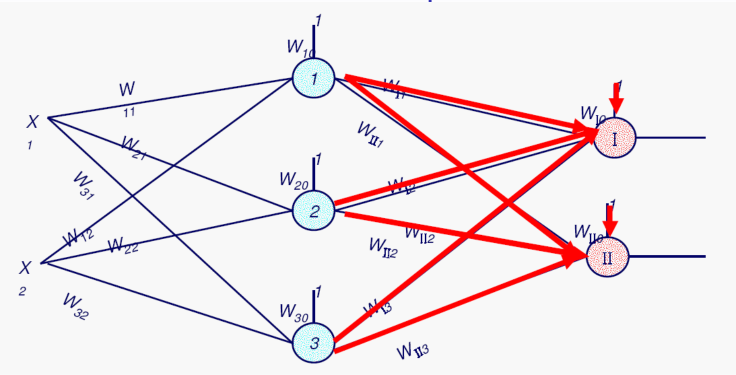


$$O_1 = O(\sum_{i=0} W_{1i} X_i) = O(W_{10}.1 + W_{11} X_1 + W_{12} X_2)$$

$$O_2 = O(\sum_{i=0} W_{2i} X_i) = O(W_{20}.1 + W_{21} X_1 + W_{22} X_2)$$

$$O_3 = O(\sum_{i=0} W_{3i} X_i) = O(W_{30}.1 + W_{31} X_1 + W_{32} X_2)$$

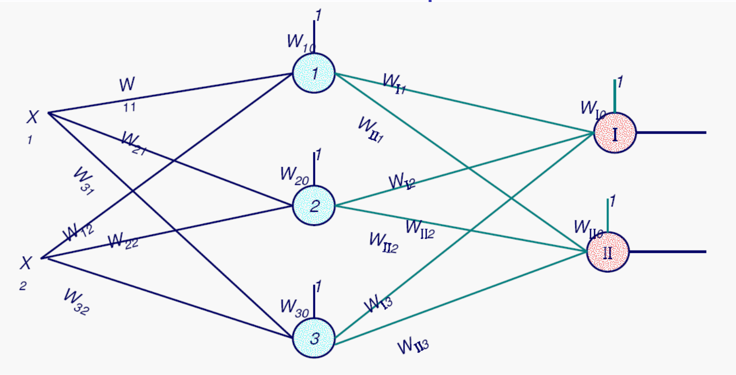
MLP example



$$O_I = O(\sum_{i=0} W_{Ii} O_i) = O(W_{I0}.1 + W_{I1}O_1 + W_{I2}O_2 + W_{I3}O_3)$$

$$O_{II} = O(\sum_{i=0} W_{IIi} O_i) = O(W_{II0}.1 + W_{II1}O_1 + W_{II2}O_2 + W_{II3}O_3)$$

MLP example

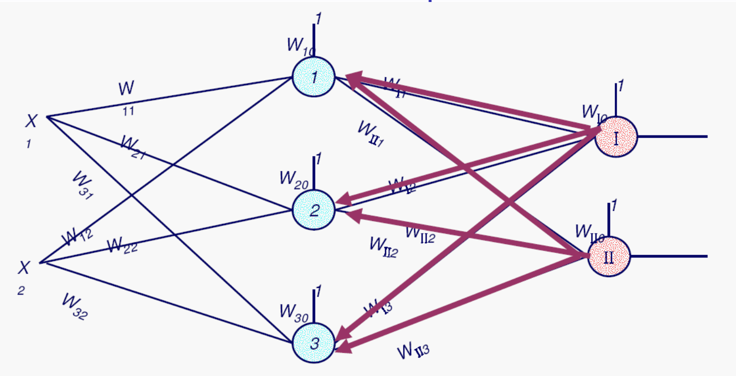


$$\Delta_I = (t_I - O_I)$$
$$\Delta_{II} = (t_{II} - O_{II})$$

$$W_{I0} = W_{I0} + \eta \Delta_I$$
$$W_{I1} = W_{I1} + \eta \Delta_I O_1$$
$$W_{I2} = W_{I2} + \eta \Delta_I O_2$$
$$W_{I3} = W_{I3} + \eta \Delta_I O_3$$

$$W_{II0} = W_{II0} + \eta \Delta_{II}$$
$$W_{II1} = W_{II1} + \eta \Delta_{II} O_1$$
$$W_{II2} = W_{II2} + \eta \Delta_{II} O_2$$
$$W_{II3} = W_{II3} + \eta \Delta_{II} O_3$$

MLP example

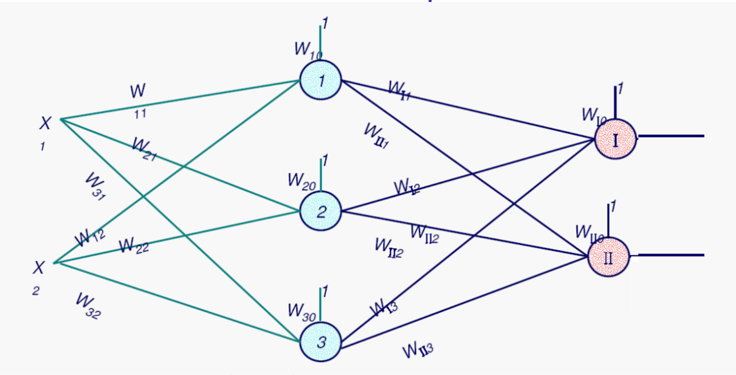


$$\Delta_1 = \sum_{k=1} W_{k1}\Delta_k O_k(1-O_k) = (W_{I1}\Delta_I O_I(1-O_I) + W_{II1}\Delta_{II} O_{II}(1-O_{II}))$$

$$\Delta_2 = \sum_{k=1} W_{k2}\Delta_k O_k(1-O_k) = (W_{I2}\Delta_I O_I(1-O_I) + W_{II2}\Delta_I) O_{II}(1-O_{II})$$

$$\Delta_3 = \sum W_{k3}\Delta_k O_k(1-O_k) = (W_{I3}\Delta_I O_I(1-O_I) + W_{II3}\Delta_I) O_{II}(1-O_{II})$$

MLP example



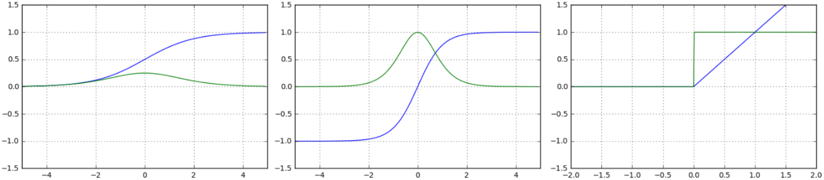
$$W_{10} = W_{10} + \eta \Delta_1 X_0 O_1 (1 - O_1)$$
$$W_{11} = W_{11} + \eta \Delta_1 X_1 O_1 (1 - O_1)$$
$$W_{12} = W_{12} + \eta \Delta_1 X_2 O_1 (1 - O_1)$$

$$W_{20} = W_{20} + \eta \Delta_2 X_0 O_2 (1 - O_2)$$
$$W_{21} = W_{21} + \eta \Delta_2 X_1 O_2 (1 - O_2)$$
$$W_{22} = W_{22} + \eta \Delta_2 X_2 O_2 (1 - O_2)$$

$$W_{30} = W_{30} + \eta \Delta_3 X_0 O_3 (1 - O_3)$$
$$W_{31} = W_{31} + \eta \Delta_3 X_3 O_3 (1 - O_1)$$
$$W_{32} = W_{32} + \eta \Delta_3 X_3 O_3 (1 - O_3)$$

learning slowdown issue for sigmoid and tangH

The derivation of sigmoid and tangH is near zero for small and large pre-activations. This slows down the learning speed.



$$\mathrm{sigm}(x) = \frac{1}{1+e^{-x}} \qquad \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \qquad \mathrm{relu}(x) = \max(0, x)$$

$$\mathrm{sigm}'(x) = \mathrm{sigm}(x)(1-\mathrm{sigm}(x)) \qquad \tanh'(x) = 1 - \tanh(x)^2 \qquad \mathrm{relu}'(x) = 1_{x>0}$$

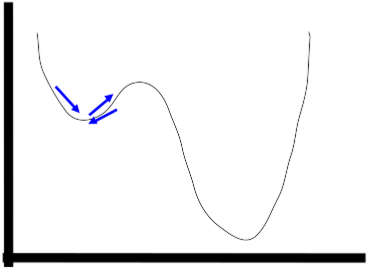
Solutions:

- Use ReLU activation function.
- Do weight and input normalization.

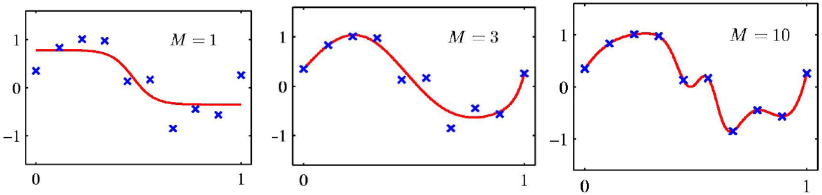
Momentum

Add a fraction (α=momentum) of the last change to the current change:

$$\Delta w_{ij} = \eta.\Delta_i.x_j.\Phi'(a_i) + \alpha.\Delta w_{ij}$$

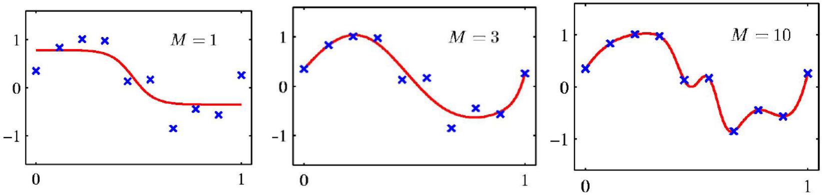


Overfitting



- **Generalization**: To establish a balance between correct responses for the training patterns and unseen new patterns.
- **Memorization**: When the model momorizes training samples instead of learning the descriptive common patterns.
- **Overfitting**: Weak generalization. It happens when the network complexity is more than the problem complexity.

Overfitting



- **Generalization**: To establish a balance between correct responses for the training patterns and unseen new patterns.
- **Memorization**: When the model momorizes training samples instead of learning the descriptive common patterns.
- **Overfitting**: Weak generalization. It happens when the network complexity is more than the problem complexity.
- In neural networks, the model complexity is specified by the number of neurons and weights.

Overfitting

- Overfitting: fitting a model to a particular training data set does not guarantee good prediction performance on unseen test data.
- If you train the net for too long, then you run the risk of overfitting.

Overfitting

- Overfitting: fitting a model to a particular training data set does not guarantee good prediction performance on unseen test data.
- If you train the net for too long, then you run the risk of overfitting.
- Lack of sufficient training data increases the risk of overfitting.

Overfitting

- Overfitting: fitting a model to a particular training data set does not guarantee good prediction performance on unseen test data.
- If you train the net for too long, then you run the risk of overfitting.
- Lack of sufficient training data increases the risk of overfitting.
- Example: consider a single neuron with 5 inputs and the following training set:

x_1	x_2	x_3	x_4	x_5	y
1	1	0	0	0	2
2	0	1	0	0	4
3	0	0	1	0	6
4	0	0	0	1	8

$$\hat{y} = \sum_{i=1}^{5} w_i \cdot x_i$$

$$\overline{W} = [2, 0, 0, 0, 0] \quad \text{Correct}$$

$$[0, 2, 4, 6, 8] \quad \text{Overfitted}$$

Avoid Overfitting

- One possible approach is to reduce the size of the network.
  - However, large networks have the potential to be more powerful than small networks.

Avoid Overfitting

- One possible approach is to reduce the size of the network.
  - However, large networks have the potential to be more powerful than small networks.
- Provide more training samples (not always possible).

Avoid Overfitting

- One possible approach is to reduce the size of the network.
  - However, large networks have the potential to be more powerful than small networks.
- Provide more training samples (not always possible).
- Stop learning before overfitting happens.

Avoid Overfitting

- One possible approach is to reduce the size of the network.
  - However, large networks have the potential to be more powerful than small networks.
- Provide more training samples (not always possible).
- Stop learning before overfitting happens.
- Use regularization terms to dynamically adjust network complexity.

Avoid Overfitting

- One possible approach is to reduce the size of the network.
  - However, large networks have the potential to be more powerful than small networks.
- Provide more training samples (not always possible).
- Stop learning before overfitting happens.
- Use regularization terms to dynamically adjust network complexity.
- Use ensemble methods.

Avoid Overfitting

- One possible approach is to reduce the size of the network.
  - However, large networks have the potential to be more powerful than small networks.
- Provide more training samples (not always possible).
- Stop learning before overfitting happens.
- Use regularization terms to dynamically adjust network complexity.
- Use ensemble methods.
- use random dropout technique for hidden neurons.

Regularization

- Since a larger number of parameters causes overfitting, a natural approach is to constrain the model to use fewer non-zero parameters.
- The most applied regularization is adding the penalty $\lambda||W||$ to the loss function .

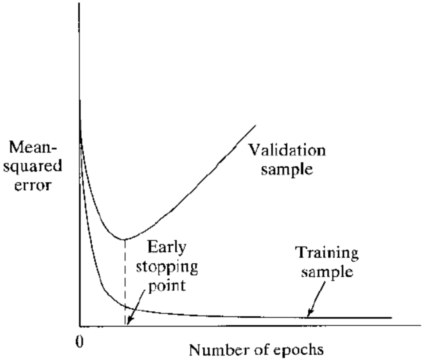
$$L = \frac{1}{2}(Y - y)^2 + \lambda||W||$$

- Therefore the learnin rule is re-written as:

$$\Delta w_{ij} = \eta.\Delta_i.x_j.\Phi'(a_i) - \eta.\lambda.w_{ij}$$

Early stopping method of training

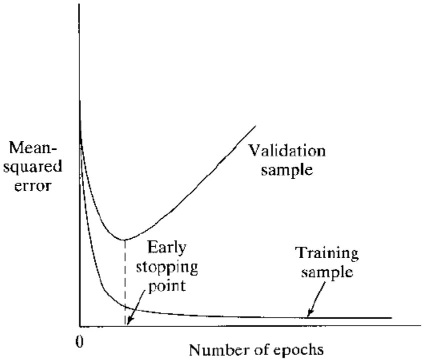
- Split training samples into a training set (80%) and a validation set (20%).



- Stop learning when the loss decreases on train set but increases on validation set.

Dropout or Dropconnect

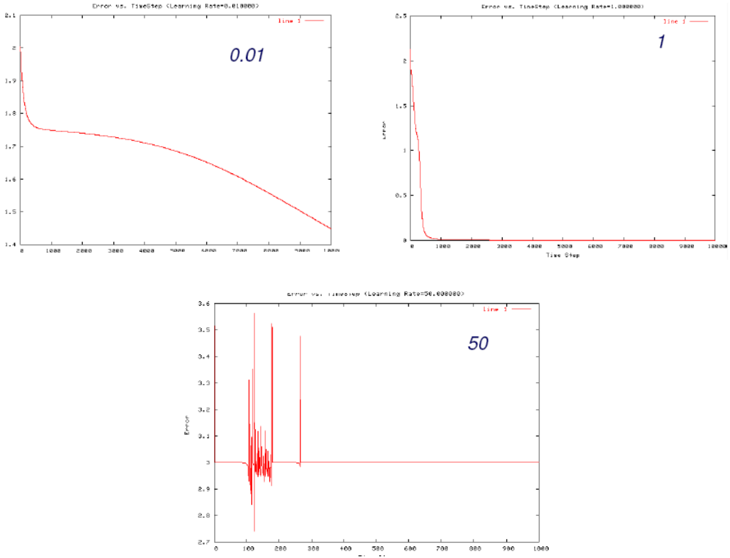
- Temporarily inactivate some of the hidden layer neurons (Dropout) or some of their weights(Dropconnect).
- These methods act like training multiple network inside one network (to decrease overfitting and increase genralization)



The Vanishing and Exploding Gradient

- While increasing depth often reduces the number of parameters, the chain rule can cause in vanishing or exploding gradients.
- A sigmoid activation often encourages the vanishing gradient problem, because its derivative is less than 0.25.
- A ReLU activation unit is known to be less likely to create a vanishing gradient problem because its derivative is always 1 for positive values of the argument.
- **Adaptive learning rates** and **conjugate gradient methods** are other solutions.

Importance of Learning Rate



Learning mode

There are two basic modes of updating weights:

- The **pattern mode** in which weights are updated after the presentation of a single training pattern,
  - It is easier to fit into memory.
  - For larger datasets it can converge faster.
  - Due to frequent updates the steps taken towards the minima of the loss function have oscillations which can help getting out of local minimums.
- The **batch mode** in which weights are updated after a batch of patterns.
  - Less oscillations and noisy steps taken towards the global minima.
  - It can benefit from vectorization which increases the speed of processing.
  - It produces a more stable gradient descent convergence.