Deep neural network

Kyung-Ah Sohn

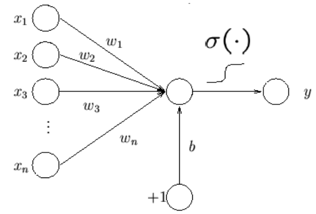
Ajou University

Contents

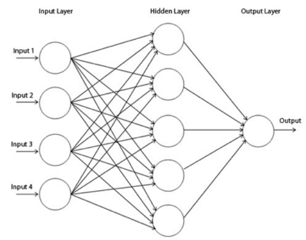
- Perceptron
- Learning XOR
- NN architecture design and computation
- Training NN: Back-propagation
- Regularization

History of neural networks

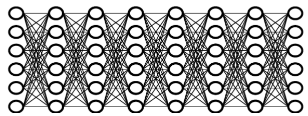
- **First generation** (1958~): perceptron (F. Rosenblatt, 1958)
  - Criticized by Marvin Minsky about XOR problem



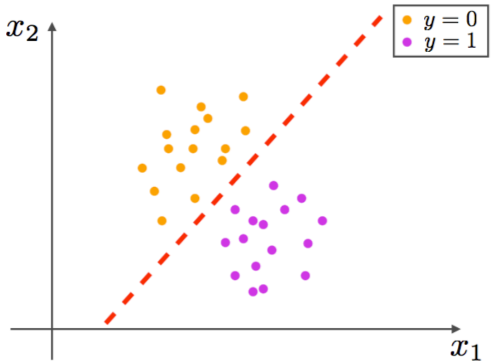
- **Second generation** (1986~) : multilayer perceptrons
  - Trained by back-propagating error signal (1986)
  - Mostly shallow network with 1 hidden layer



- **Third generation** (2006~): deep learning
  - Deep belief nets (Hinton, 2006)
  - Deep neural network (DNN), convolutional neural network (CNN), …



Recall: Logistic regression



$\mathbf{x} = (x_1, x_2)$

Model ***linear*** decision boundary

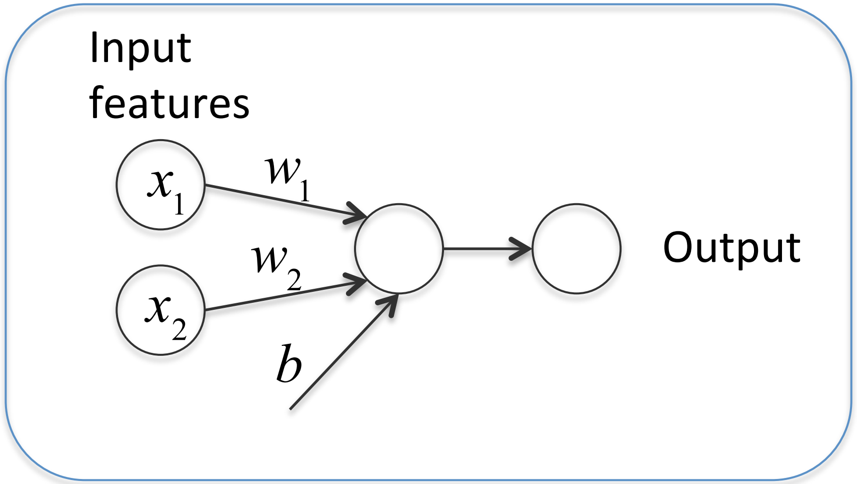
$$\log \frac{p(y=1|\mathbf{x})}{1-p(y=1|\mathbf{x})} = \mathbf{w}^\intercal \mathbf{x} + b$$

$$\rightarrow \quad p(y=1|\mathbf{x}) = \frac{1}{1+e^{-(\mathbf{w}^\intercal \mathbf{x}+b)}}$$

Input features			Output

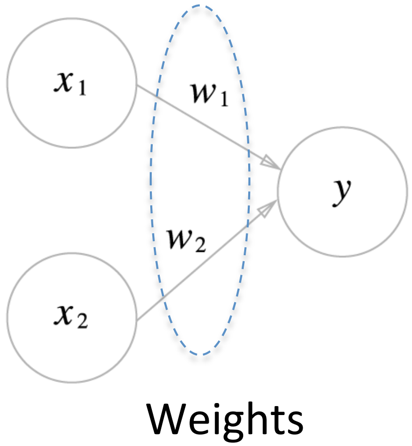
$$\boxed{(x_1, x_2)} \Rightarrow \boxed{t = w_1 x_1 + w_2 x_2 + b} \Rightarrow \boxed{g(t) = \frac{1}{1+e^{-t}}}$$

$\uparrow$ $w = (w_1, w_2), b$ $\qquad$ $\uparrow$ $g(.)$



Perceptron

- Perceptron with 2 input features

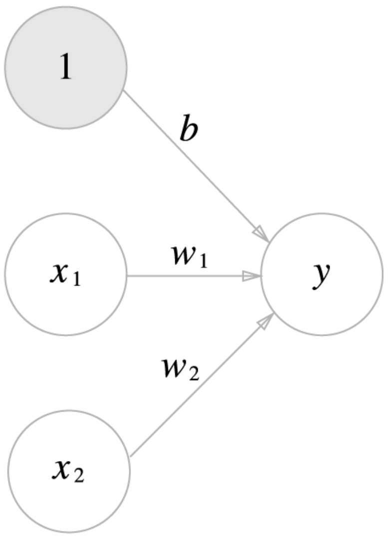


$$y = \begin{cases} 0 & (w_1 x_1 + w_2 x_2 \leq \theta) \\ 1 & (w_1 x_1 + w_2 x_2 > \theta) \end{cases}$$

threshold

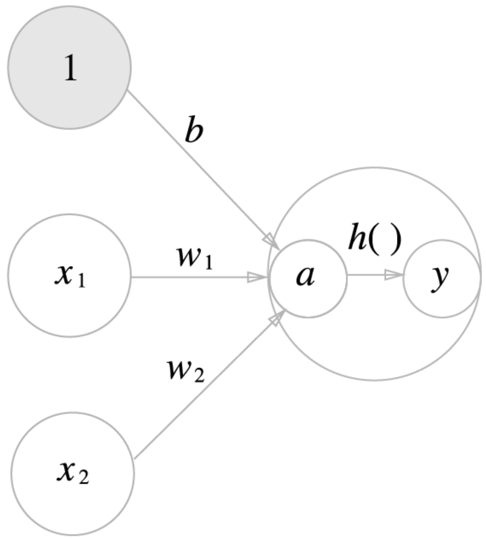
Perceptron

Representation with bias

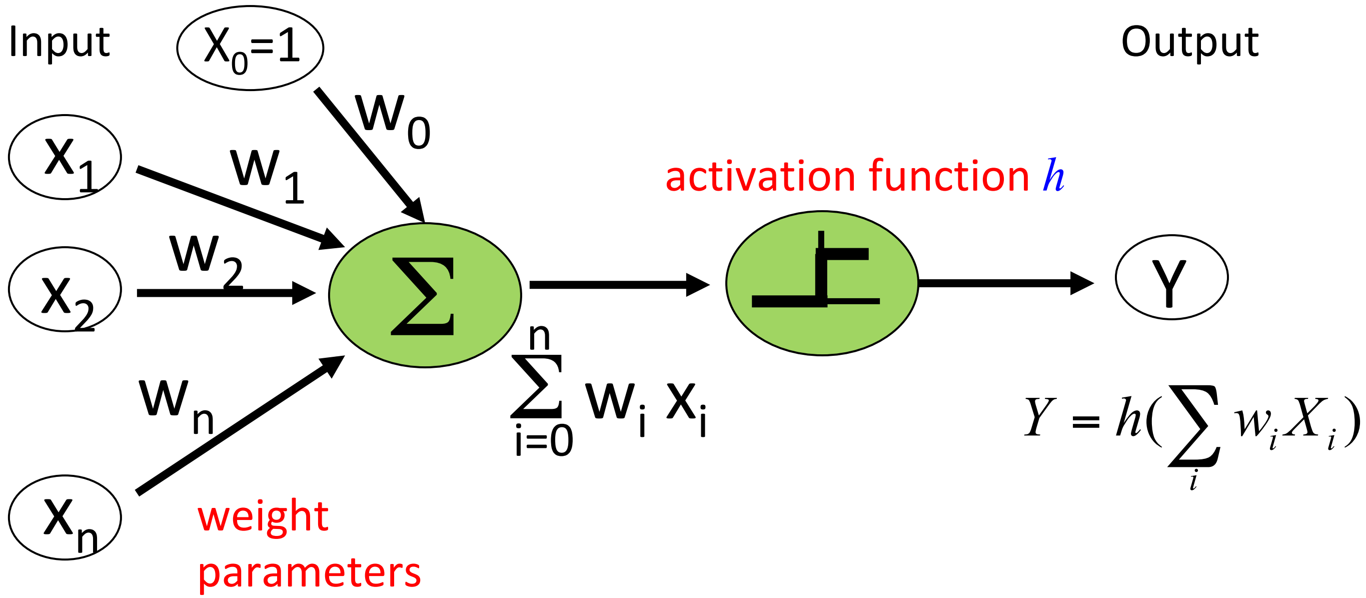


$$y = h(b + w_1 x_1 + w_2 x_2)$$

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$$

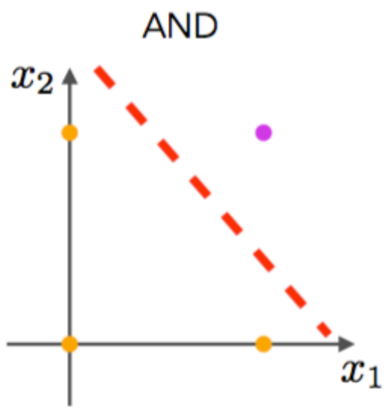


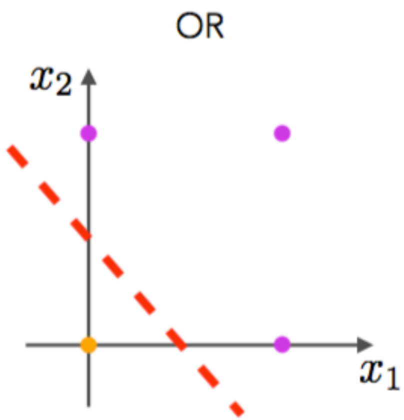
Perceptron


Input
$X_0=1$
w_0
x_1
w_1
x_2
w_2
w_n
x_n
weight parameters
$\sum_{i=0}^{n} w_i x_i$
activation function h
Output
Y
$Y = h(\sum_i w_i X_i)$

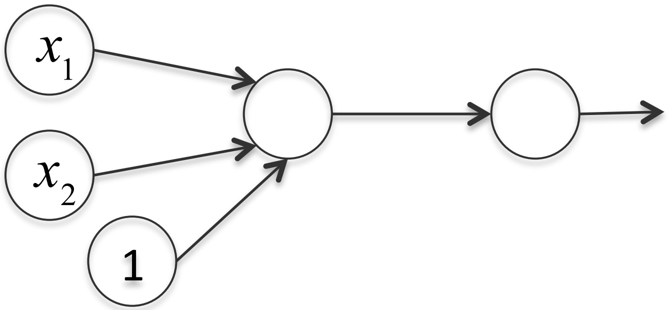
Perceptron: Boolean operation

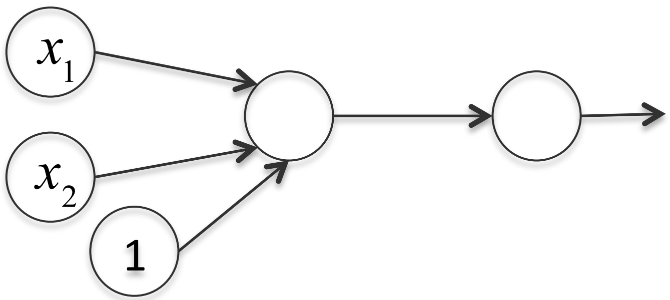
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1





x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

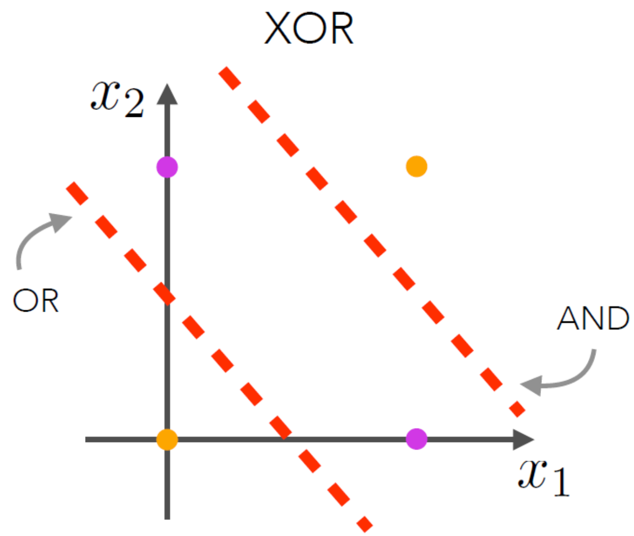




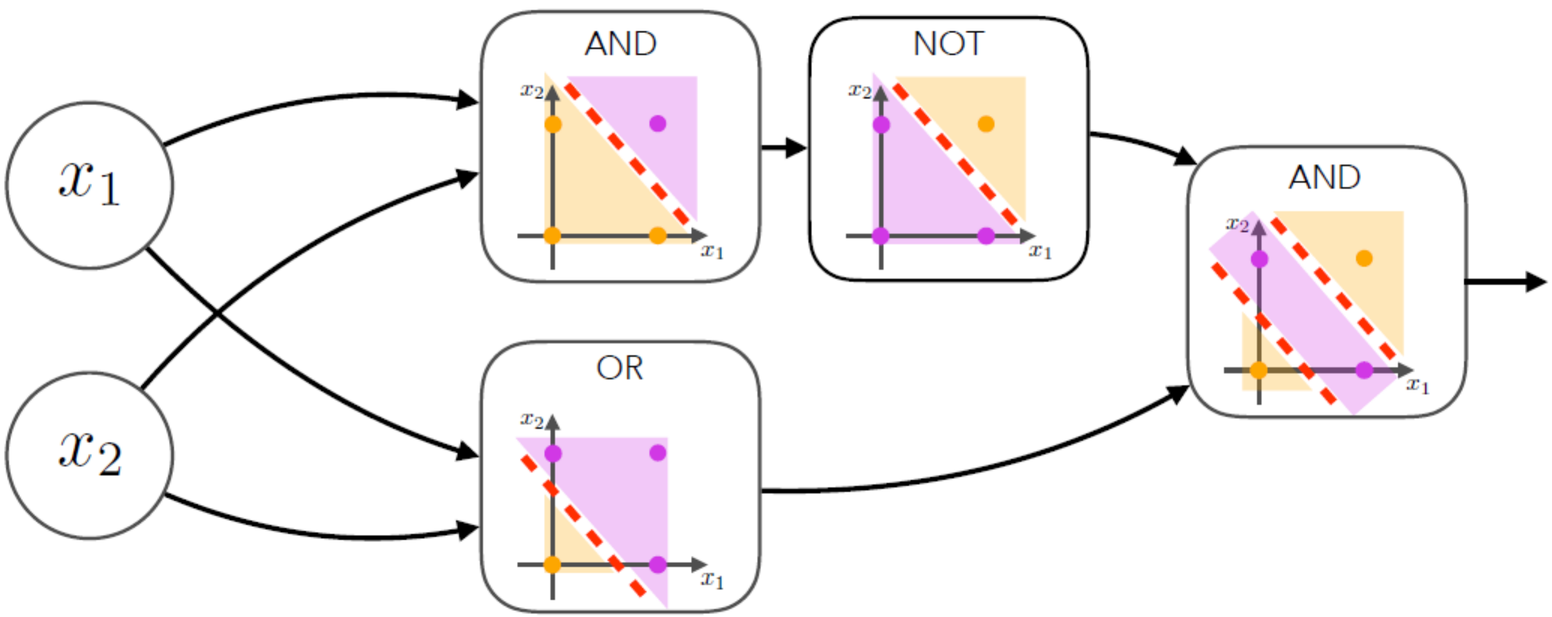
XOR is not linearly separable

Cannot be solved by a simple perceptron
But can be separated using AND and OR

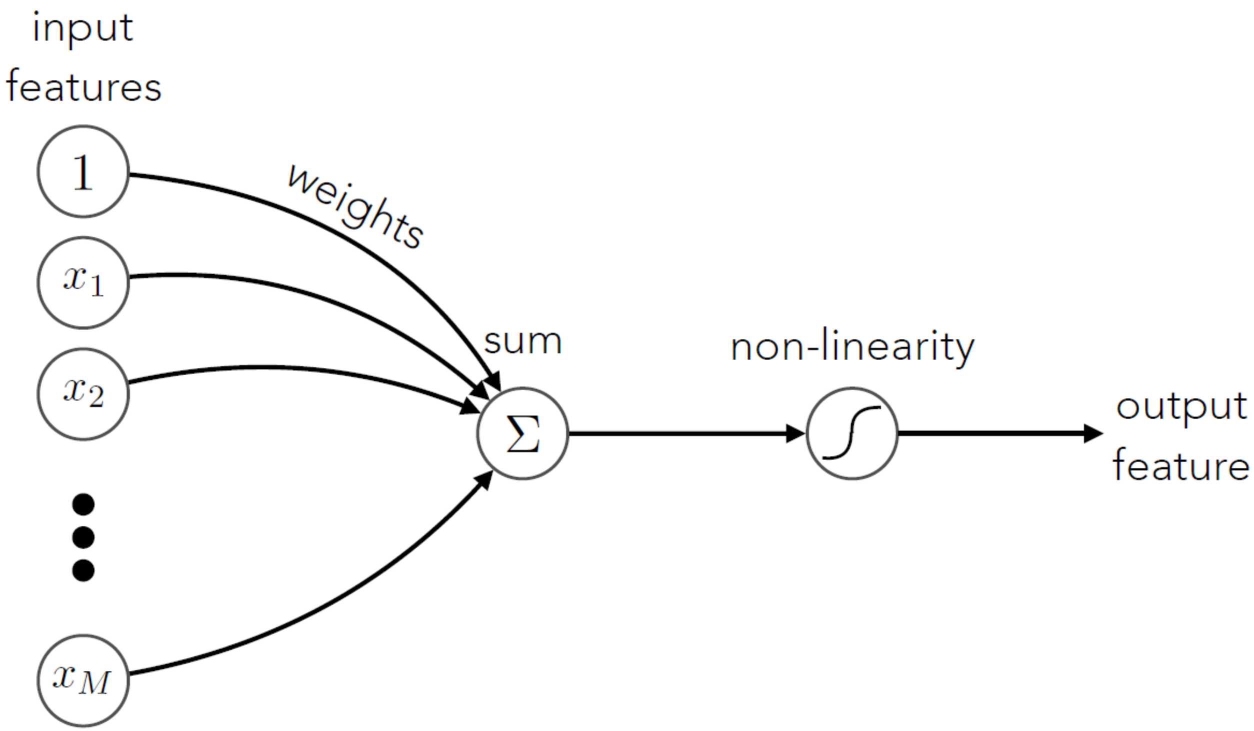
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



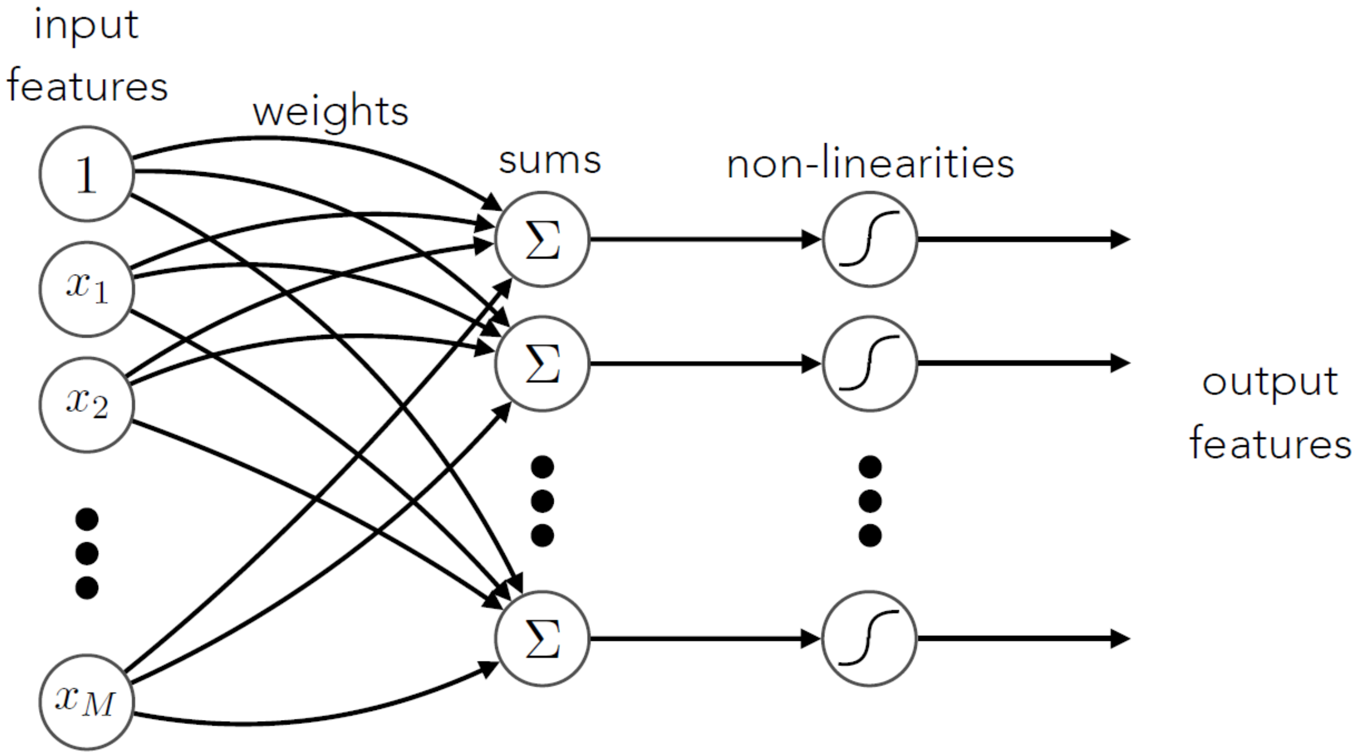
Building XOR from AND and OR



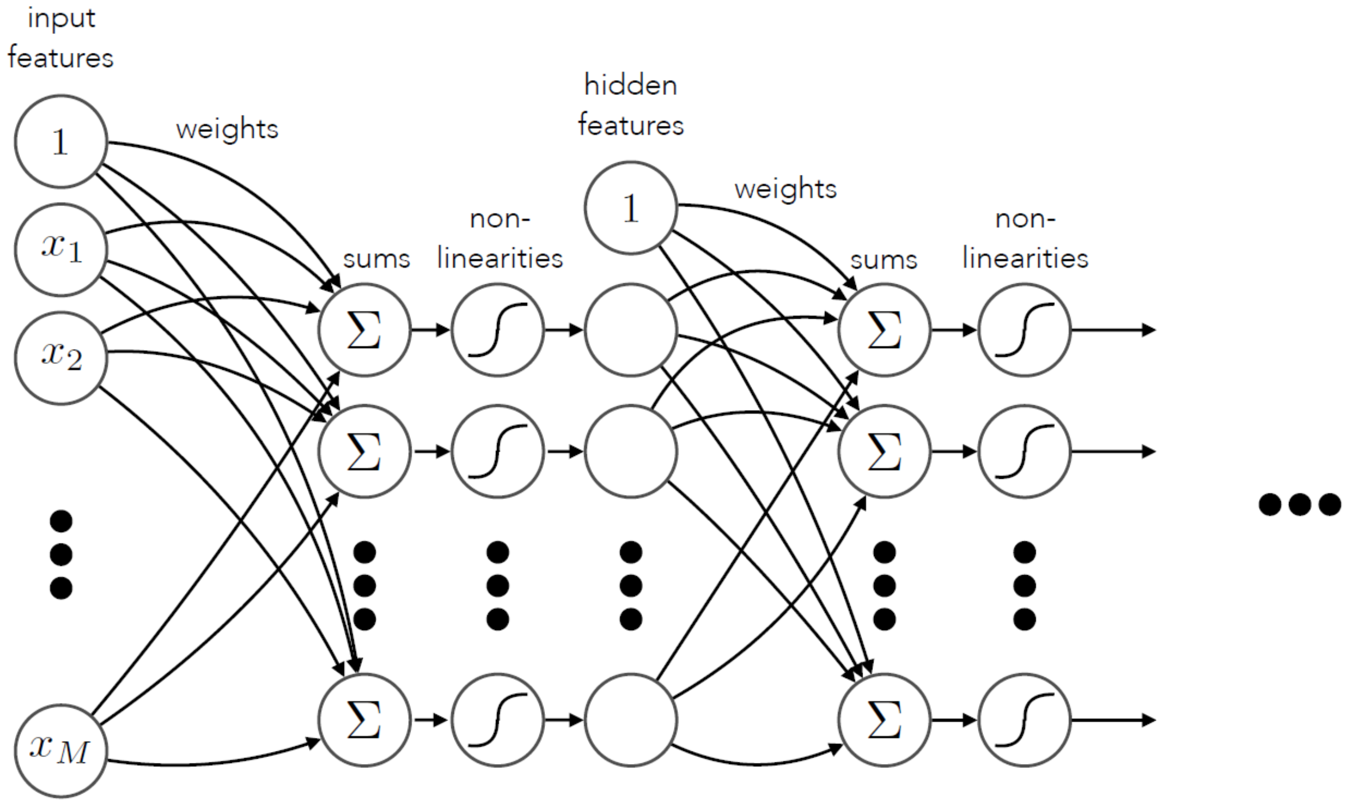
Artificial neuron



Multiple neurons form a **layer**



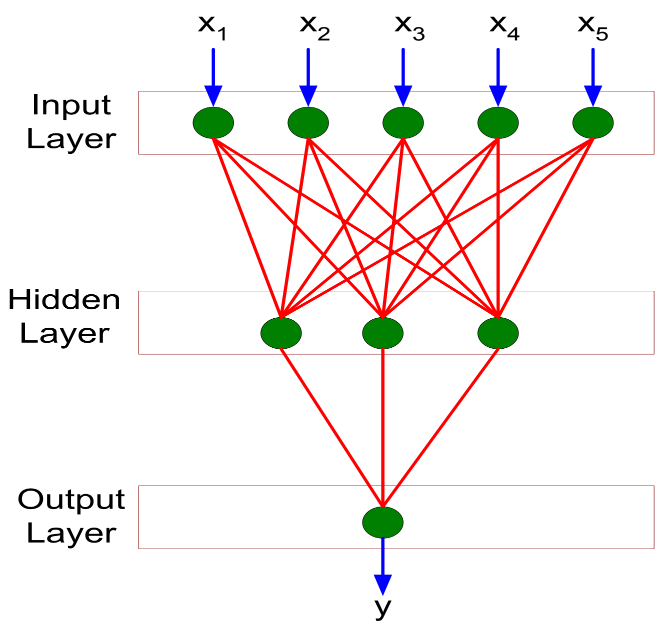
Multiple layers form a **network**

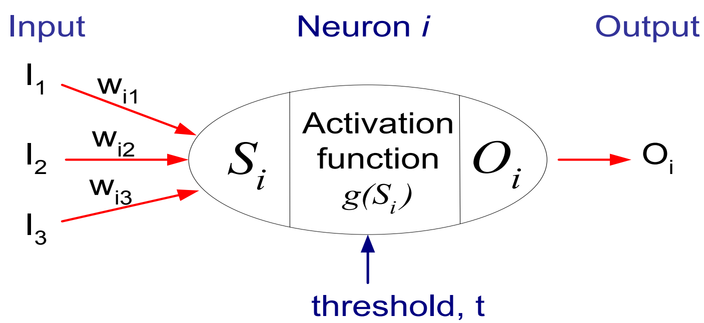


Artificial neural network

General Structure

Multilayer perceptron



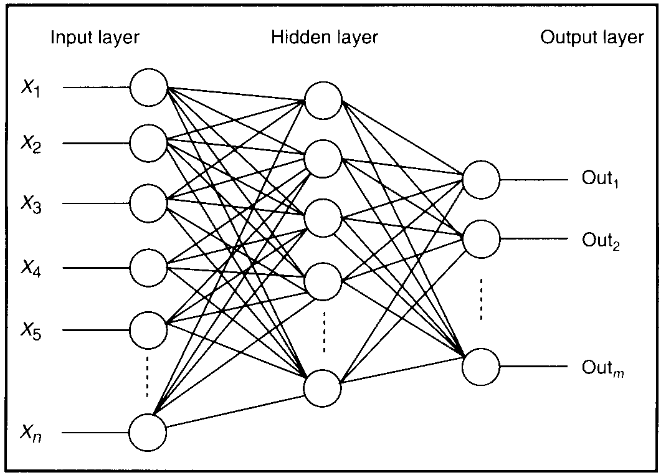


We can form non-linear functions by composing stages of processing

Training ANN means learning the weights of the neurons

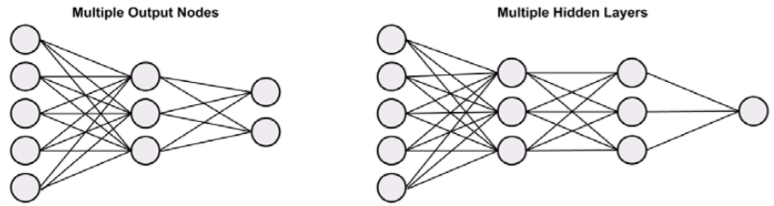
Network topology

- The number of layers
- The number of nodes within each layer
- Whether information in the network is allowed to travel backward

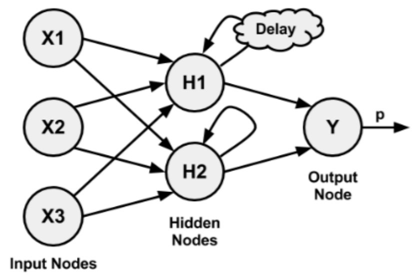


Direction of information travel

- Feed-forward network
  - Signal is fed in one direction



- Recurrent network (or feedback network)
  - Allows signal to travel in both directions using loops
  - Addition of a short term memory

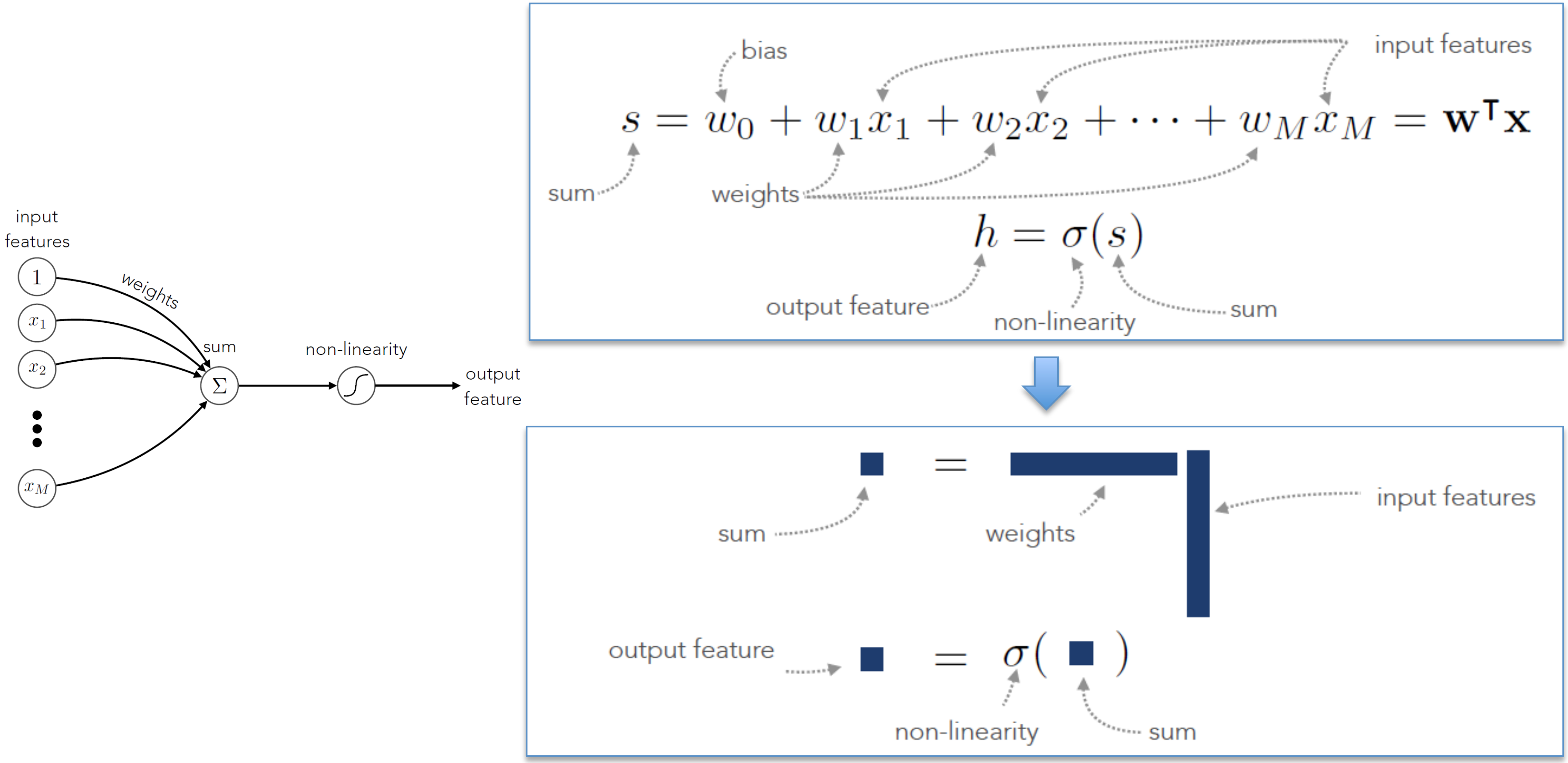


Neural network mathematics

Neural network: input / output transformation

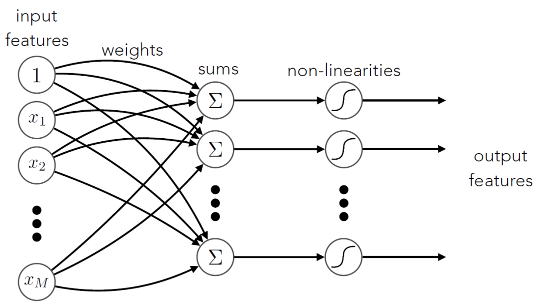
$$y_{out} = F(x, W)$$

W is the matrix of all weight vectors.



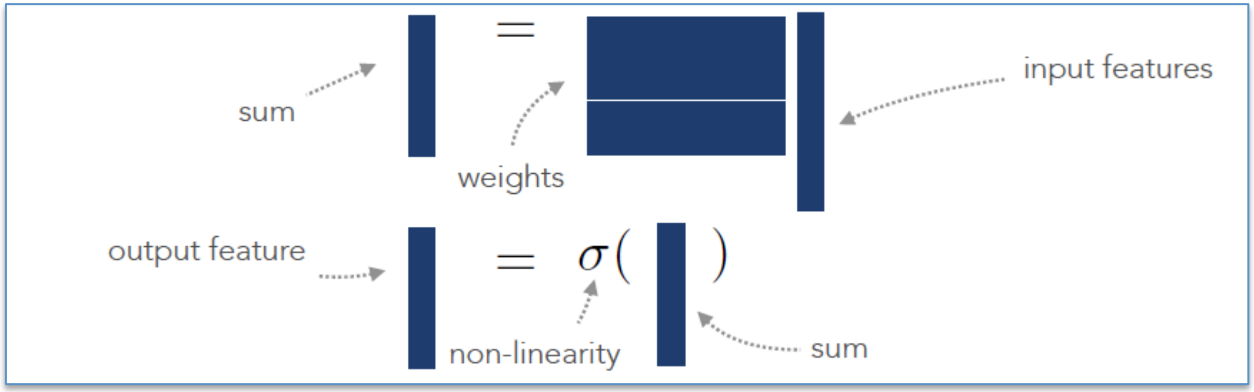
input features
1
x_1
x_2
x_M
weights
sum
non-linearity
output feature
bias
input features
$s = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_M x_M = \mathbf{w}^\mathsf{T}\mathbf{x}$
sum
weights
$h = \sigma(s)$
output feature
non-linearity
sum
sum
weights
input features
output feature
$\sigma($ $)$
non-linearity
sum

Layer: parallelized weighted sum and non-linearity

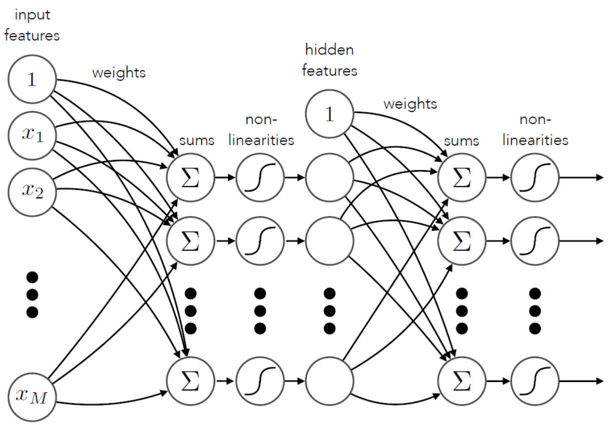


one sum per weight *vector* $s_j = \mathbf{w}_j^\intercal \mathbf{x}$ $\longrightarrow$ $\mathbf{s} = \mathbf{W}^\intercal \mathbf{x}$ vector of sums from weight *matrix*

$$\mathbf{h} = \sigma(\mathbf{s})$$



Network: sequence of parallelized weighted sums and non-linearities



•••

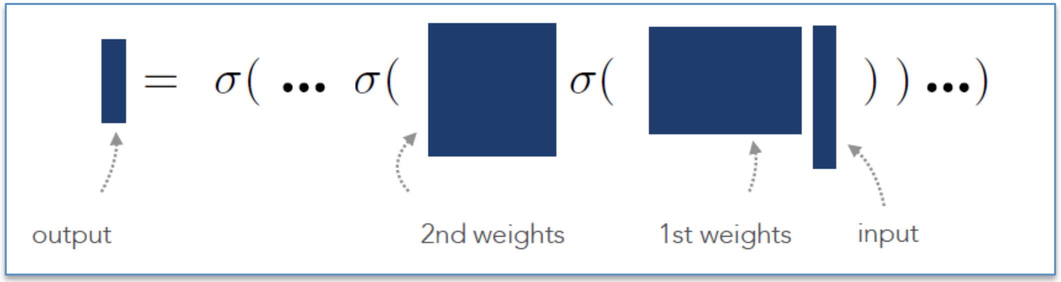
1st layer

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$$
$$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$$

2nd layer

$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$$
$$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$$

•••

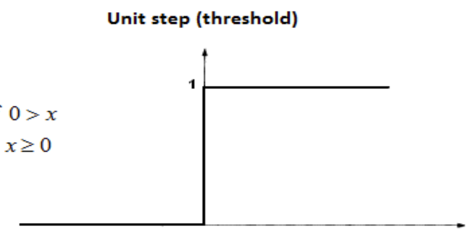


Activation functions

- The mechanism by which the artificial neuron processes information and passes it throughout the network
- Create non-linearity

- Unit step function (threshold activation function)
    - Not continuous
    - Returns either 0 or 1 (discrete)
    - Non-linear function

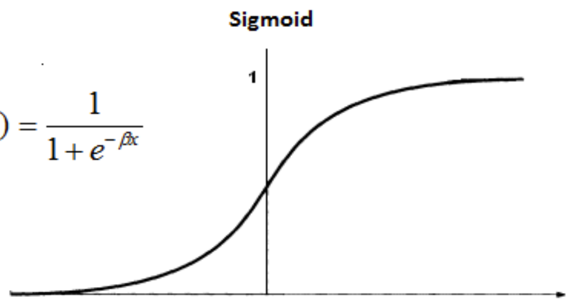
$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$



Activation functions

- Sigmoid function
  - The values of logistic function range from 0 to 1 (continuous)
  - Differentiable (mathematically)
  - Non-linear
  - The most commonly used in traditional ANN

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



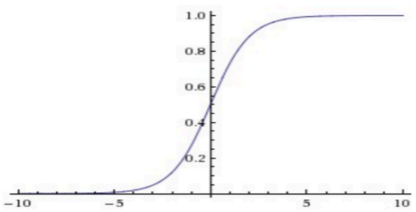
Activation functions

- tanh: takes a real-valued input and squashes it to the range [-1, 1]

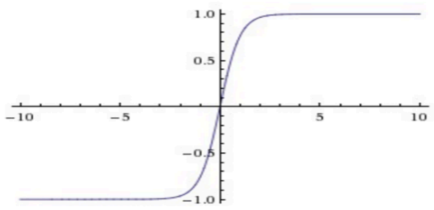
- ReLU: ReLU stands for Rectified Linear Unit. It takes a real-valued input and thresholds it at zero (replace negative values with zero)

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

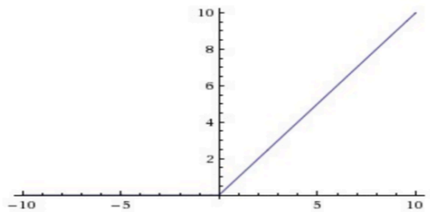
$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$



Sigmoid



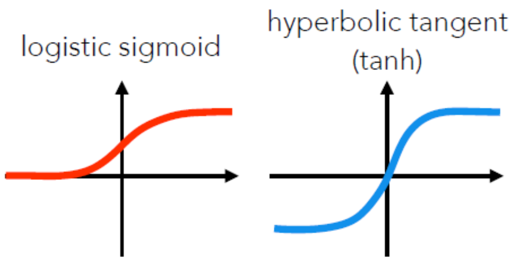
tanh



ReLU

Activation functions

- Traditionally

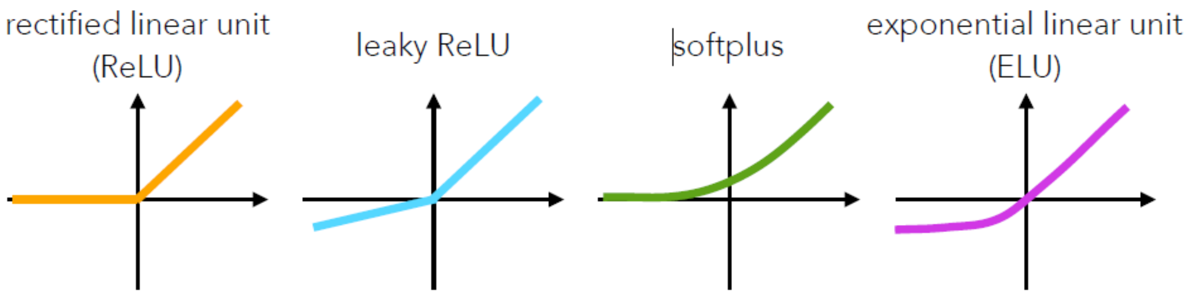


saturating

derivative goes to

<u>zero</u> at +∞ and -∞

- More recently



non-saturating

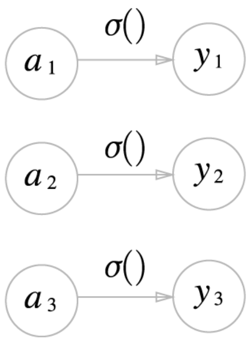
<u>non-zero</u> derivative

at +∞ and/or -∞

Activation at output layer

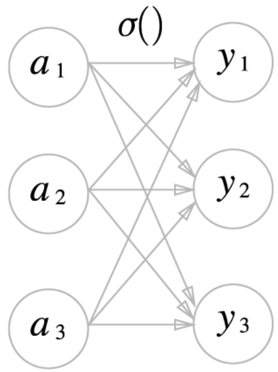
For regression

- Identity function



For classification

- Softmax function



$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

Loss function
ex: MNIST classification

- Measure to calculate how 'bad' the NN is
- e.g. MNIST classification

```
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

: Output from NN
: True label

  – Mean squared error (MSE)

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$

  – Cross entropy

$$E = -\sum_{k} t_k \log y_k$$

Loss function

- Suppose the neural network's computed outputs and the target values are as follows

NN1

```
computed       | targets              | correct?
-----------------------------------------------
0.3  0.3  0.4  | 0   0   1 (democrat)   | yes
0.3  0.4  0.3  | 0   1   0 (republican) | yes
0.1  0.2  0.7  | 1   0   0 (other)      | no
```

NN2

```
computed       | targets              | correct?
-----------------------------------------------
0.1  0.2  0.7  | 0   0   1 (democrat)   | yes
0.1  0.7  0.2  | 0   1   0 (republican) | yes
0.3  0.4  0.3  | 1   0   0 (other)      | no
```

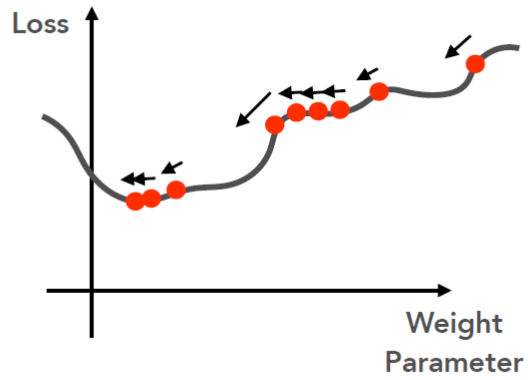
	NN1	NN2
Classification error	0.33	0.33
Avg. Cross-Entropy	1st sample: -(0*log0.3+0*log0.3+1*log0.4)=-log0.4 ➔ ACE=(-log0.4-log0.4-log0.1)/3=1.38	0.64
MSE	(0.54+0.54+1.34)/3=0.81	0.34

Learning as optimization

- To learn the weights, we need the **derivative** of the loss w.r.t the weights

  How should the weight be updated to decrease the loss?

$$w = w - \alpha \frac{\partial \mathcal{L}}{\partial w}$$



With multiple weights

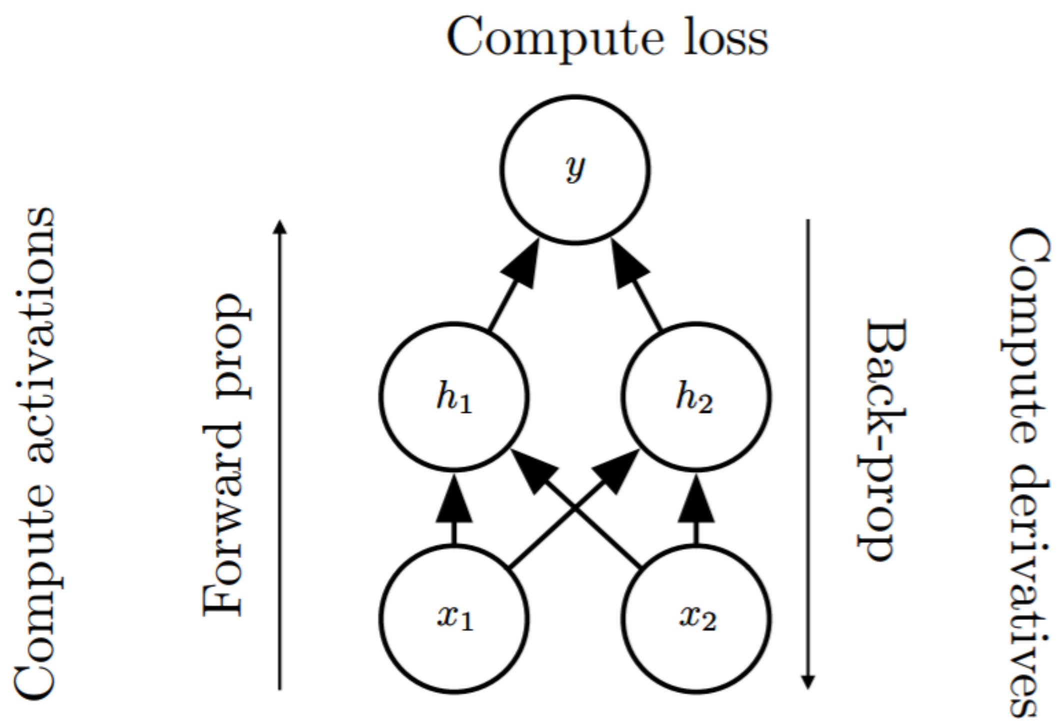
$$\mathbf{w} = \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}$$

gradients

Learning rates

Stochastic Gradient Descent (SGD)

Simple Back-Propagation example



Backpropagation

A neural net defines a function of composed operations

$$f_L(\mathbf{w}_L, f_{L-1}(\mathbf{w}_{L-1}, \ldots f_1(\mathbf{w}_1, \mathbf{x}) \ldots))$$

and the loss *L* is a function of the network output

→ Use chain rule to calculate gradients

chain rule example

$$y = w_2 e^{w_1 x}$$

input x output y parameters w_1, w_2

evaluate parameter derivatives: $\frac{\partial y}{\partial w_1}, \frac{\partial y}{\partial w_2}$

define

$$v \equiv e^{w_1 x} \longrightarrow y = w_2 v$$

$$u \equiv w_1 x \longrightarrow v = e^u$$

then

$$\frac{\partial y}{\partial w_2} = v = e^{w_1 x}$$

chain rule

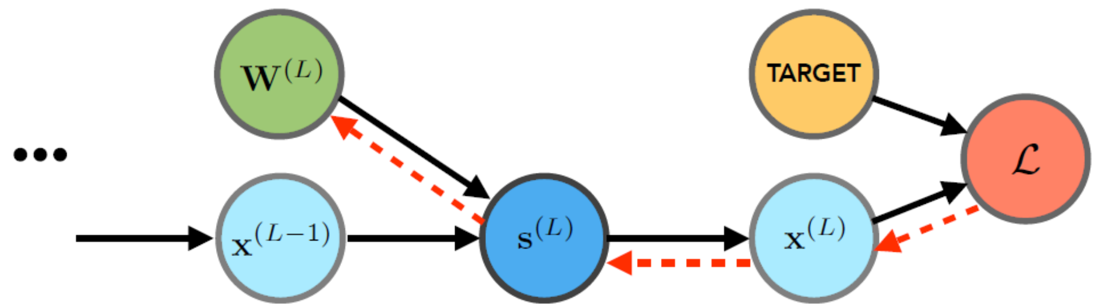
$$\frac{\partial y}{\partial w_1} = \boxed{\frac{\partial y}{\partial v} \frac{\partial v}{\partial u} \frac{\partial u}{\partial w_1}} = w_2 \cdot e^{w_1 x} \cdot x$$

Backpropagation

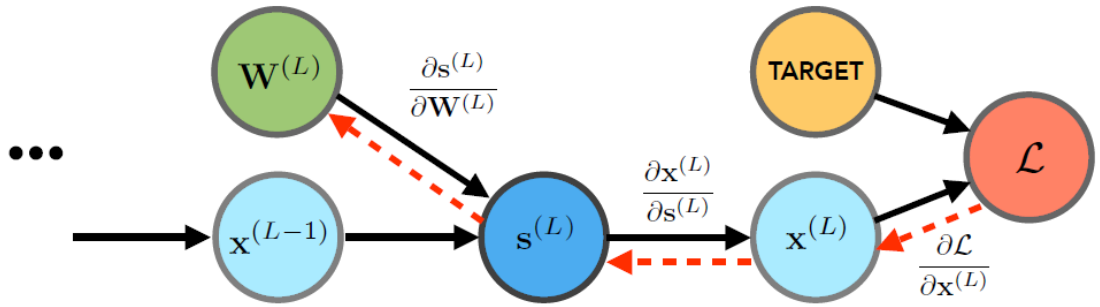
- Recall

1st layer	2nd layer		Loss
$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$	$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$	$\bullet\bullet\bullet$	$\mathcal{L}$
$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$	$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$		

- To determine the chain rule ordering, we draw the dependency graph



Backpropagation



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

depends on the form of the loss

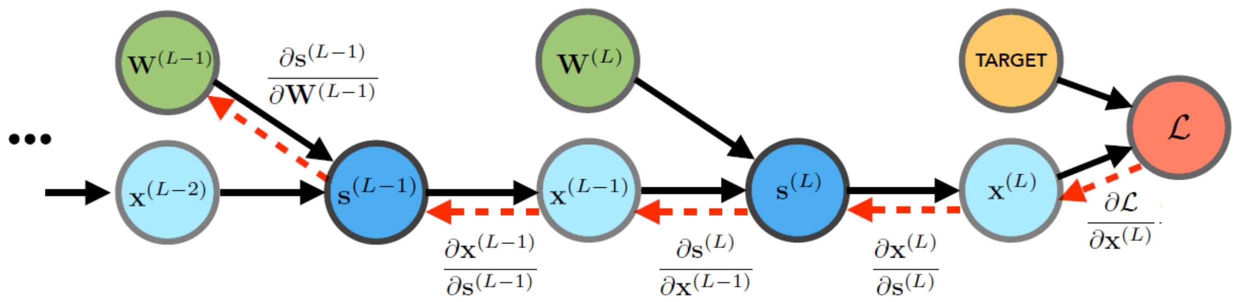
derivative of the non-linearity

$$\frac{\partial}{\partial \mathbf{W}^{(L)}}(\mathbf{W}^{(L)\intercal}\mathbf{x}^{(L-1)})$$
$$= \mathbf{x}^{(L-1)\intercal}$$

note $\nabla_{\mathbf{W}^{(L)}}\mathcal{L} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$ is notational convention

Backpropagation

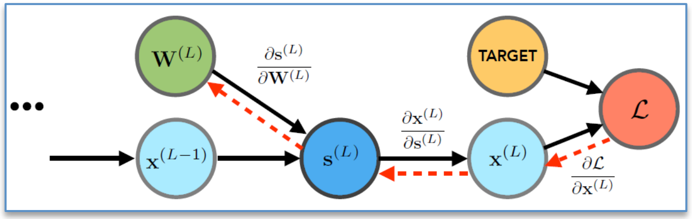
- Go back one more layer



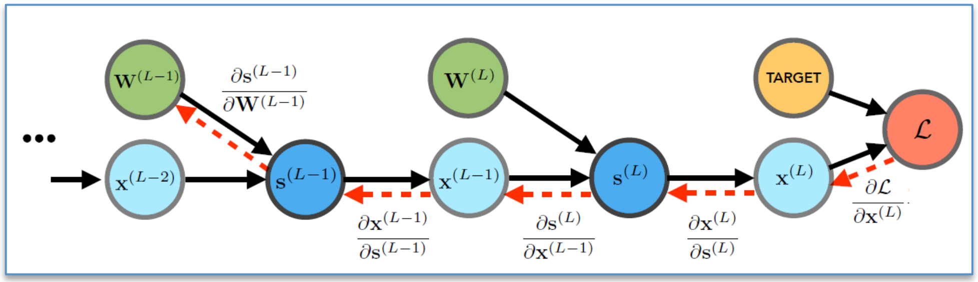
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L\text{-}1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial \mathbf{s}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$$

Backpropagation

- Some of the terms appear in both gradients
- e.g. we can reuse $\dfrac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell)}}$



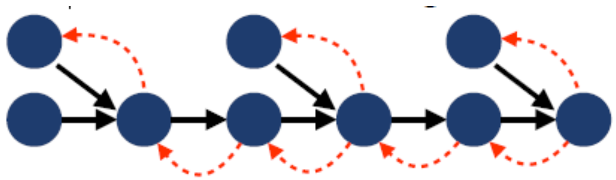
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \underline{\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$$



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L-1)}} = \underline{\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial \mathbf{s}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$$

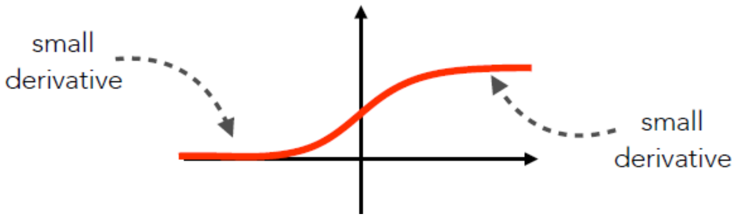
Backpropagation

- Update weights using gradients
- BP calculates the gradients via chain rule
- Gradient is propagated backward through the network

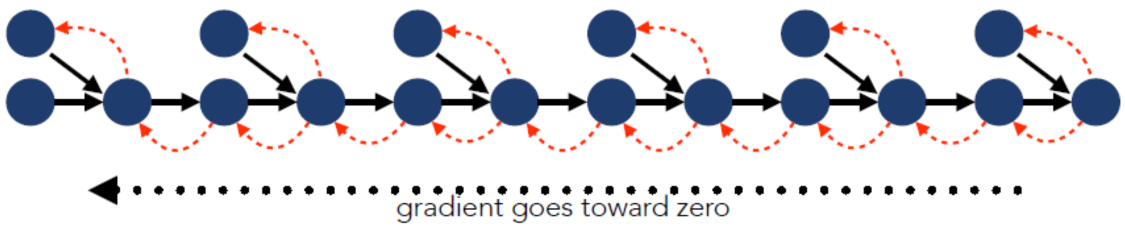


- Most deep learning software libraries automatically calculate gradients

Vanishing gradient

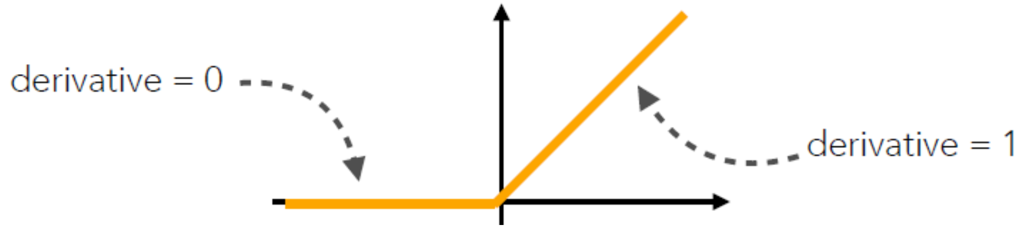


Small (~0) derivatives almost everywhere



- In backprop, the product of many small terms goes to zero
- *Difficult to train very deep neural network with sigmoid*

ReLU



- In the positive region, ReLU does not saturate, preventing gradients from vanishing in deep networks

- In the negative region, ReLU saturates at zero, resulting in ‘dead units’, but in practice, this doesn’t seem to be a problem

- Most commonly used in DNN

Training NN: summary

- 전제
  - 신경망의 가중치(weights)와 편향(bias)을 훈련데이터에 적응하도록 조정하는 과정을 ‘학습 ' 이라고 함

- 1단계- 미니배치
  - 훈련데이터 중 일부를 무작위로 가져옴. 이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실(Loss) 값을 줄이는 것이 목표
- 2단계 – Gradient (기울기) 산출
  - 미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수(weight parameter)의 기울기를 구함. 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시함
- 3단계 – 매개변수 갱신
  - 가중치 매개변수를 기울기 방향으로 아주 조금 갱신함
- 4단계 – 반복
  - 1~3단계를 반복함.

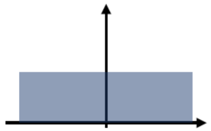
$$E = \frac{1}{N}\sum_{t=1}^{N}(F(x_t;W) - y_t)^2$$

$$\Delta w_i^j = -c \cdot \frac{\partial E}{\partial w_i^j}(W)$$

$$w_i^{j,new} = w_i^j + \Delta w_i^j$$

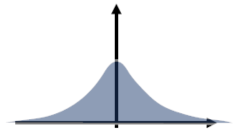
Initialization

- Learning can be sensitive to weight <u>initialization</u>



$w \sim U(-a, a)$

uniform



$w \sim \mathcal{N}(0, \sigma)$

Gaussian

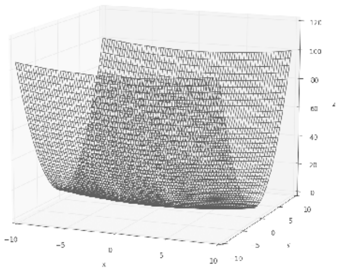
- *Xavier* initialization (2010)
- *He* initialization (2015)

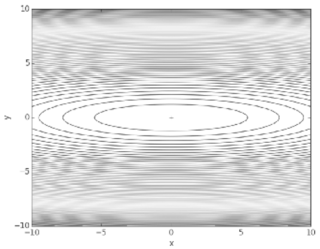
Parameter update

- SGD (확률적 경사 하강법)
  - Simple, easy to implement
  - <u>*Inefficient*</u> sometimes

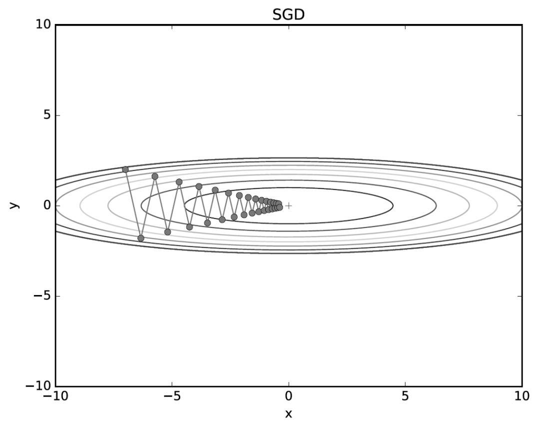
$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$f(x,y) = \frac{1}{20}x^2 + y^2$$





Optimization path



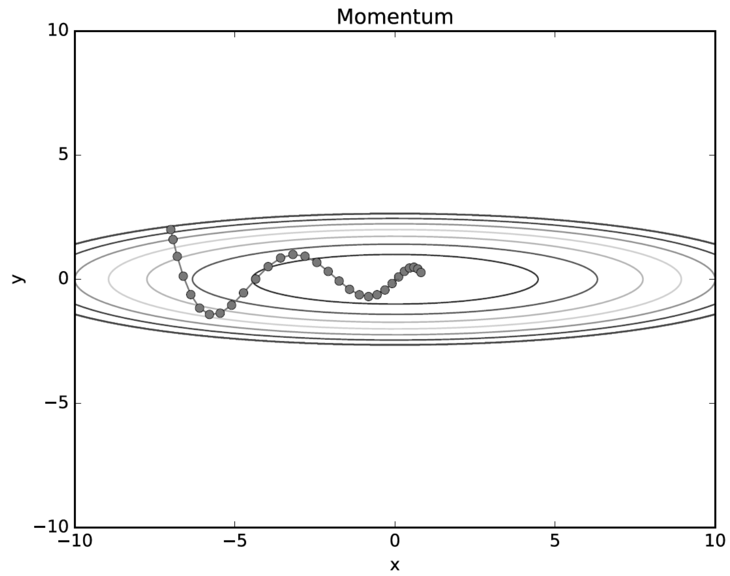
Parameter update

- Momentum (운동량)

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

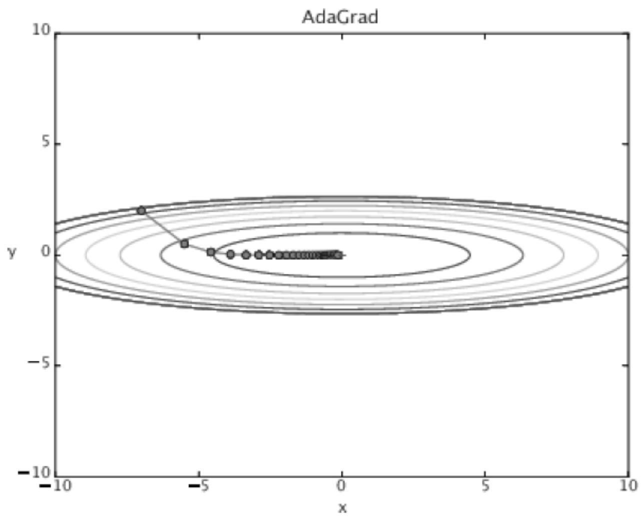
$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

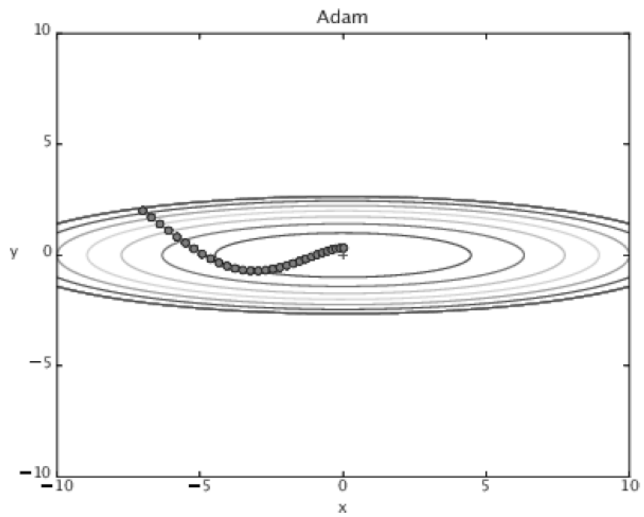
Optimization update path by momentum



Parameter update

AdaGrad **Adam**





RMSprop, Adadelta, Adamax, …

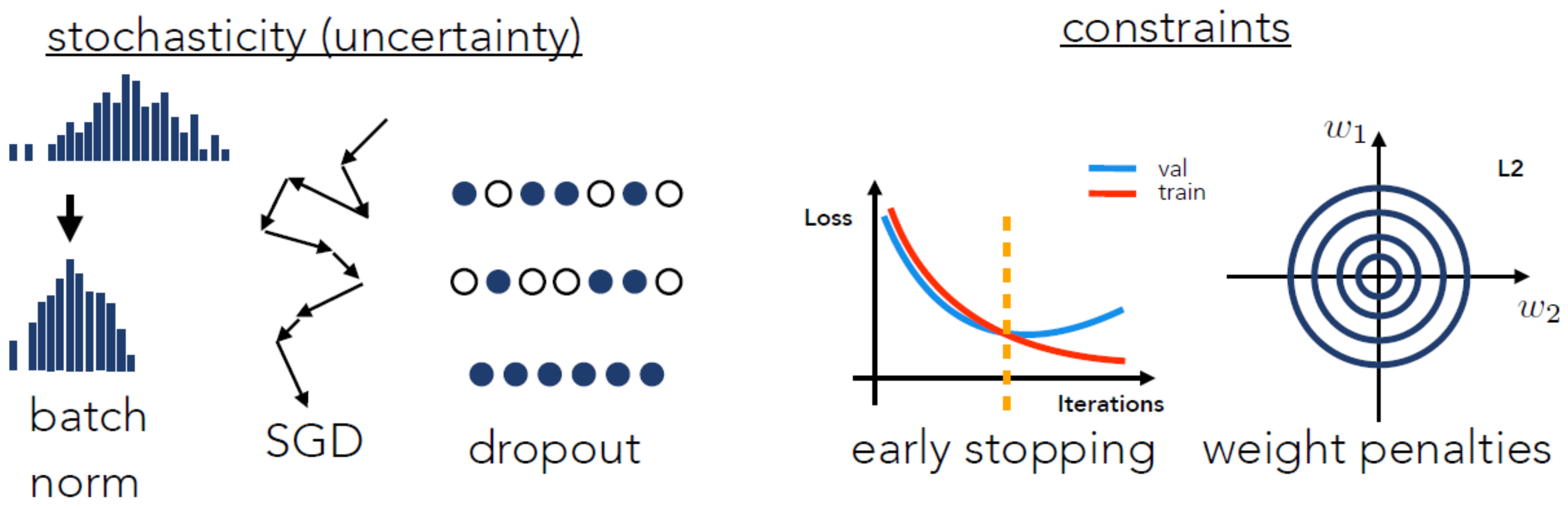
Overfitting

- Especially with
  - Complex models with many parameters
  - Small training data
- How to avoid
  - Weight decay (add L2 penalty on W to loss)
  - Dropout: randomly *drop*(remove) neurons during training

    Regularization

Regularization

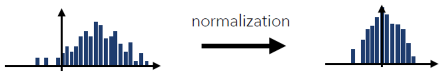
- Regularization combats overfitting



Batch normalization

- Splits the training dataset into small mini-**batches** that are used to calculate model error and update model coefficients.
- **Normalization** transform distribution into standard Normal

$$X_{\text{normal}} = \frac{X_{\text{original}} - \mu}{\sigma}$$

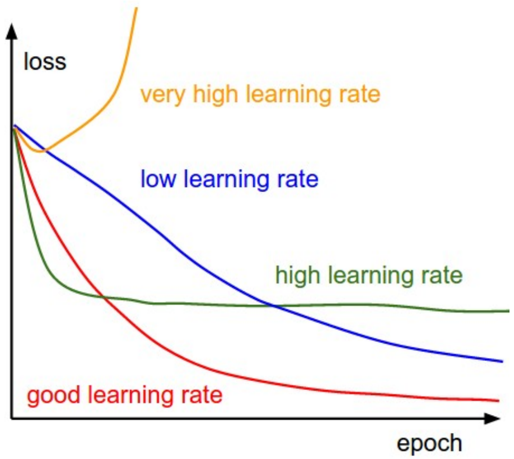


- **Batch norm**. normalizes each layer´s activations according to the statistics of the batch
  – Results less sensitive to initialization

$$\mathbf{s}^{(\ell)} \leftarrow \gamma \frac{\mathbf{s}^{(\ell)} - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}} + \beta$$

Hyperparameter tuning

- Hyperparameters
  - Number of nodes, Batch size
  - Learning rate (initial, decay schedule), regularization strength (L2 penalty, dropout)
  - Tuned over validation set

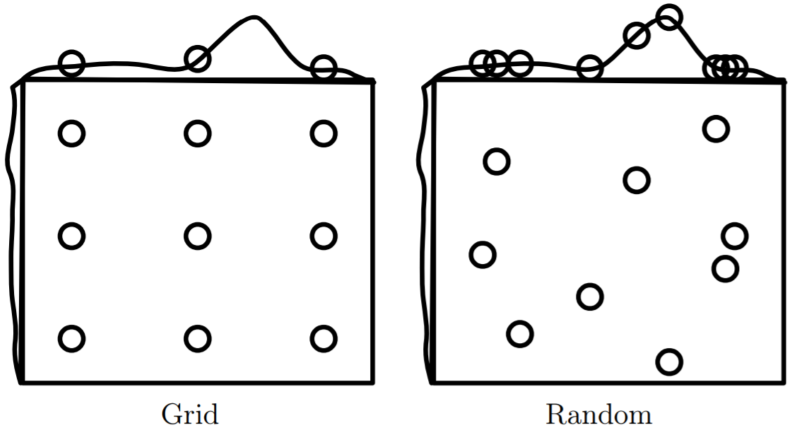


Hyperparameter tuning

- Search on log scale

```
learning_rate = 10 ** uniform(-6, 1)
```

- Random search vs. grid search



- From coarse to fine ranges

Learning automatically from data