# Neural Networks

## 4. Multi-layer Perceptron & Back-propagation

Farkaš, Kuzma et al.

Center for Cognitive Science
Department of Applied Informatics
Faculty of Mathematics, Physics and Informatics UKBA

March 13th, 2018

# Conventions

# Conventions, vol. 1

$c$ – scalar

$\mathbf{x}$ – column vector ("western" algebra)
$x_i$ – $i$-th element of a vector (scalar)

$\mathbf{W}$ – matrix
$\mathbf{w}_i$ – $i$-th row of matrix (vector)
$w_{i,j}$ – element in the $i$-th row and $j$-th column (scalar)

# Conventions, vol. 2

$c\mathbf{x}$, $c\mathbf{W}$ – scalar/elementwise multiplication – shape is preserved

$\mathbf{WX}$ – matrix multiplication
$\mathbf{Wx}$ – matrix-vector multiplication, i.e. applying transformation $\mathbf{W}$ to a vector $\mathbf{x}$

$\mathbf{xy}$ – mistake
$\mathbf{x}^T\mathbf{y} = \mathbf{x} \cdot \mathbf{y}$ – scalar/dot/inner product – result is a scalar
$\mathbf{x} \times \mathbf{y}$ – vector product – result is a vector
$\mathbf{xy}^T$ – outer product – result is a matrix

# Numpy

# Numpy

- `np.zero`, `np.one` `np.eye`
- uniform `np.random.rand`, normal/gaussian `np.random.randn`
- `+`, `-`, `*`, `np.sin(…)`, `np.exp(…)` all work elementwise
- `+=`, `-=`, `*=` also work
- `np.dot(a,b)` = `a.dot(b)` = `a @ b`
- `np.transpose(a)` = `a.transpose()` = `a.T`
- transpose on a vector does nothing!
    - use `np.inner` or `np.outer`
    - alternatively, `np.atleast_2d(x).T` returns a matrix posing as a row vector
- `np.linalg.norm`
- `np.min/max/mean/std(x, axis=0)`

# Multi-layer Perceptron

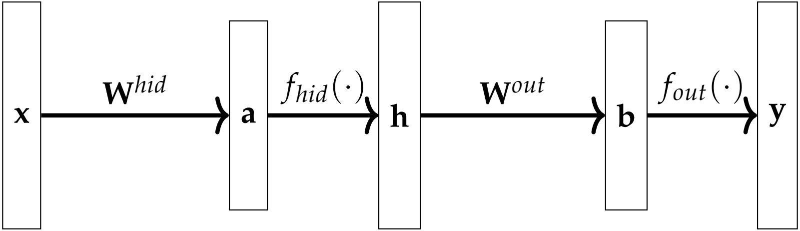
# Multi-layer Perceptron

$$\mathbf{x} \xrightarrow{\mathbf{W}^{hid}} \mathbf{a} \xrightarrow{f_{hid}(\cdot)} \mathbf{h} \xrightarrow{\mathbf{W}^{out}} \mathbf{b} \xrightarrow{f_{out}(\cdot)} \mathbf{y}$$

$\mathbf{x}$ $\qquad$ $\mathbf{h} = f_{hid}(\mathbf{W}^{hid}\mathbf{x})$ $\qquad$ $\mathbf{y} = f_{out}(\mathbf{W}^{out}\mathbf{h})$

- virtual input for the bias term: $x_{n+1} = h_{m+1} = 1$

# Activation functions




- logistic sigmoid:
    - $\mathrm{logsig}(x) = \frac{1}{1+e^{-x}}$
    - $\mathrm{logsig}'(x) = \mathrm{logsig}(x)(1 - \mathrm{logsig}(x))$
- hyperbolic tangent
- rectified linear units (ReLU)
- linear (makes sense only on output)
- ...

# Back-propagation

# Back-propagation

$$\mathbf{x} \xrightarrow{\mathbf{W}^{hid}} \mathbf{a} \xrightarrow{f_{hid}(\cdot)} \mathbf{h} \xrightarrow{\mathbf{W}^{out}} \mathbf{b} \xrightarrow{f_{out}(\cdot)} \mathbf{y}$$

- $g_i^{out} = (d_i - y_i) f'_{out}(b_i)$
- $g_k^{hid} = \left(\sum_i w_{i,k}^{out} g_i^{out}\right) f'_{hid}(a_k)$
- $\Delta \mathbf{W}^{out} = \mathbf{g}^{out} \mathbf{h}^T \qquad \mathbf{W}^{out}(t+1) = \mathbf{W}^{out}(t) + \alpha \Delta \mathbf{W}^{out}(t)$
- $\Delta \mathbf{W}^{hid} = \mathbf{g}^{hid} \mathbf{x}^T \qquad \mathbf{W}^{hid}(t+1) = \mathbf{W}^{hid}(t) + \alpha \Delta \mathbf{W}^{hid}(t)$

# Algorithm

## Initialization:

1. choose model parameters (# of hidden neurons)
2. choose training parameters (learning rate, #epochs)
3. generate random initial weights

## Training:

- until stopping criterion (accuracy / #epochs / time...):
  - with each training sample $(\mathbf{x}, \mathbf{d})$ in random order:
    - forward-pass: compute $\mathbf{h}$, $\mathbf{y}$
    - backward-pass: compute $\mathbf{g}^{out}, \mathbf{g}^{hid}$
    - adjust weights $\mathbf{W}^{hid}, \mathbf{W}^{out}$