ASSIGNMENT 7: FIRST NEURAL NETWORKS AND A GLIMPSE AT PYTORCH

Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Copyright statement:

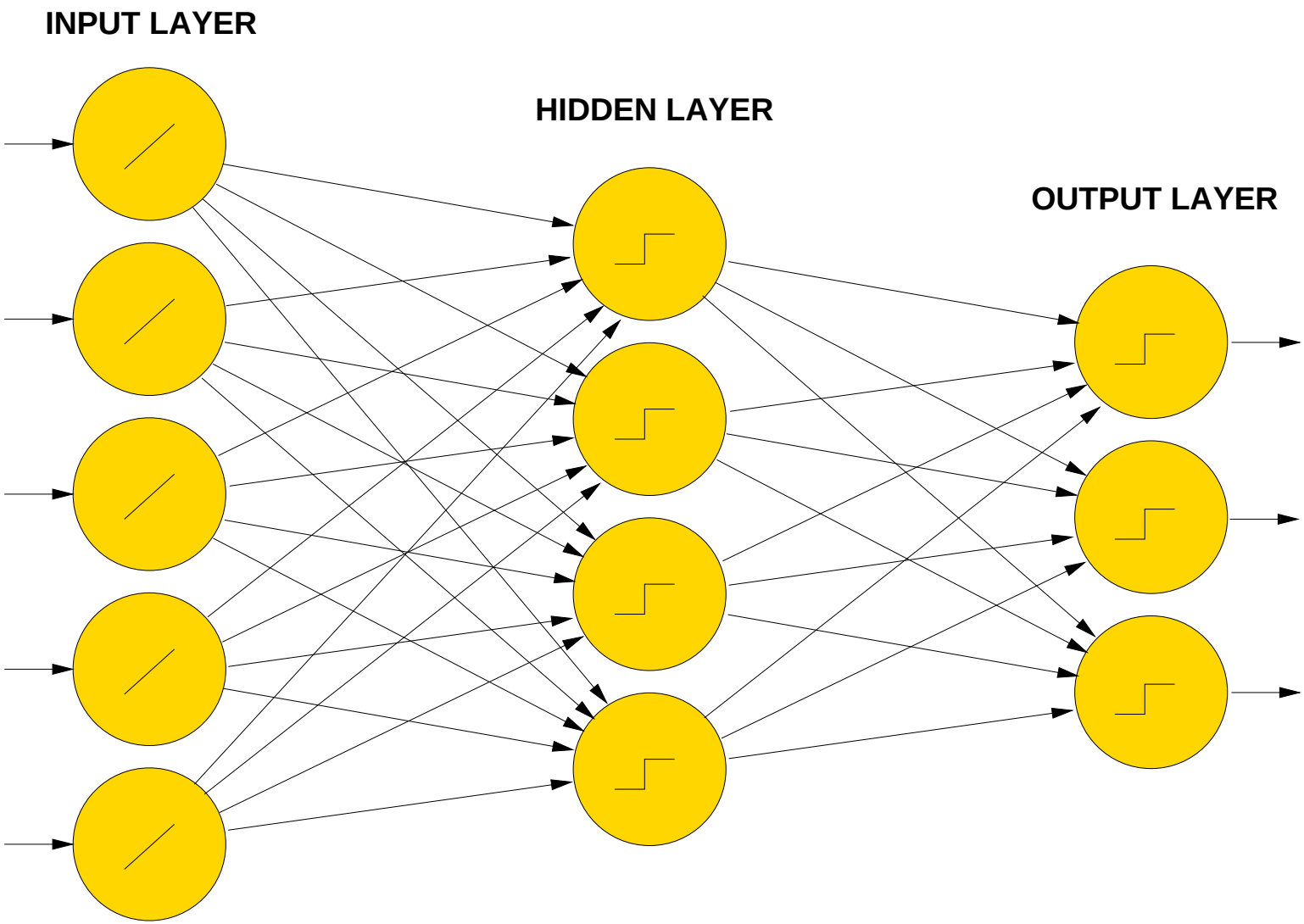
Perceptrons

- Introduced by Frank Rosenblatt in 1958
- A perceptron is a simple linear threshold unit:

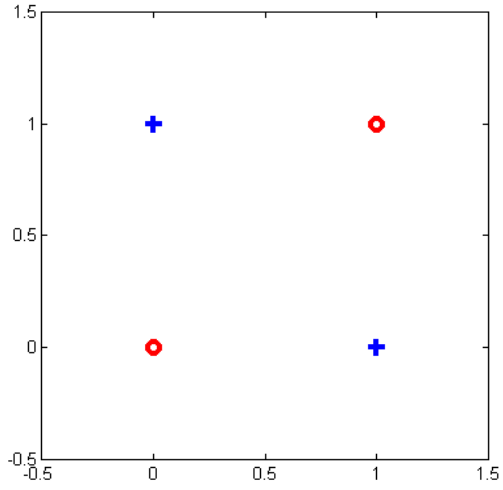
$$g(\mathbf{x}_i; \mathbf{w}, \theta) = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x}_i > \theta \\ 0 & \text{otherwise} \end{cases}$$

- In analogy to the biological model:
  - inputs $\mathbf{x}_i \Rightarrow$ charges received from connected cells
  - weights $\mathbf{w} \Rightarrow$ properties of the synaptic interface
  - ouput $\Rightarrow$ impulse that is sent through the axon as soon as the charge exceeds the threshold θ
- Though it seems to be a (simplistic) model of a neuron, a perceptron is nothing else but a simple linear classifier.

Multi-layer perceptrons



Learning non-linear functions



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

This classification problem:

- ... can't be solved by one-layer network (single-layer perceptron, logistic regression)
- ... is trivial to solve using an MLP.

How powerful are neural networks?

- Useful measure of a machine learning algorithm: what are the most complicated functions it can learn?
- Measured by the VC dimension[1]
- NNs can learn (much) more complex decision functions.
- Neural Networks are "Universal Function Approximators".

[1] **V**apnik-**C**hervonenkis dimension – to learn more, see "Statistical Learning Theory" (Vladimir N. Vapnik)

NNs: Basic definitions and notation: Part 1

- a_i: activity of the i-th unit ($a_0 = 1$)
- w_{ij}: weight from unit j to unit i ($w_{i0} = b_i$ called bias weight)
- W: number of weights
- Q: number of units
- D: number of input units ($1 \leq i \leq D$) located in the first layer called input layer.
- K: number of output units ($Q - K + 1 \leq i \leq Q$) located in the last layer called output layer.
- H: number of hidden units ($D < i \leq Q - K$) located in the hidden layers.
- L: number of layers, where L_ν is the index set of the ν-th layer; $L_1 = \{1, \ldots, D\}$ and $L_L = \{Q - K + 1, \ldots, Q\}$.
- s_i: network input or pre-activation of the i-th unit ($i > D$): $s_i = \sum_{j=0}^{Q} w_{ij} a_j$

NNs: Basic definitions and notation: Part 2

- f: activation function with $a_i = f(s_i)$ It is possible to define different activation functions f_i for different units.
- Architecture of a NN is given through:
  - number of layers
  - number of units in the layers
  - connections between units
  - also the activation function may be accounted to architecture
- A feed-forward NN has only connections from units in lower layers to units in higher layers:

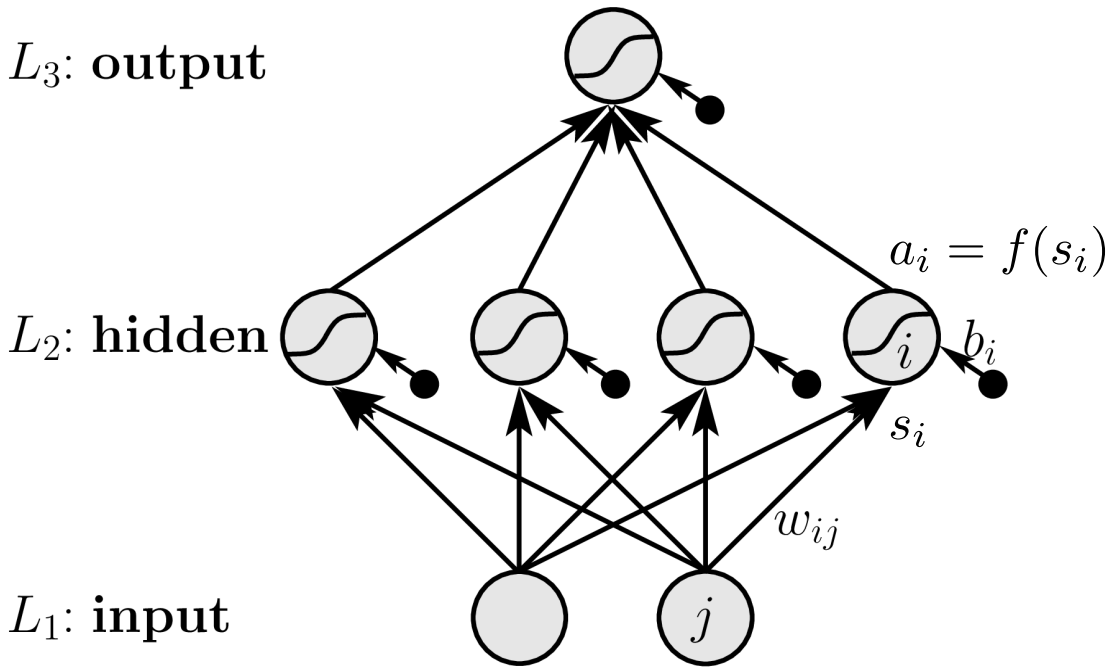
$$j \in L_\nu \text{ and } i \in L_{\nu'} \text{ and } \nu' \leq \nu \Rightarrow w_{ij} = 0.$$

- For conventional NN: only connections or weights between consecutive layers. Other weights are fixed to zero.
- Connections between units in layers which are not adjacent are called shortcut connections.

NNs: Forward pass

1. Provide input $\boldsymbol{x}$
2. Input layer $\nu = 1$: for $i = 1$ to D: $a_i = x_i$
3. Further layers: for $\nu = 2$ to L
   - For i in L_ν:
     - $s_i = \sum_{j=0}^{Q} w_{ij}\, a_j$
     - $a_i = f(s_i)$
4. Output layer: for all $Q - K + 1 \leq i \leq Q$:
   - provide output $\hat{y}_i = g(\boldsymbol{x}_i; \boldsymbol{w}) = a_i$

NNs: Visualization



Back-propagation

Calculating gradients: Back-propagation Part 1

- Similar as in Unit 4 and Unit 5 (logistic regression and gradient boosting): Expected Risk Minimization (ERM) by gradient descent (GD).
- Gradient of NN can be computed efficiently by back-propagation. Popular since mid 80ies.
- Empirical risk:

$$R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{Y}) = \frac{1}{N} \sum_{n=1}^{N} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w})).$$

  $\boldsymbol{w}$ now contains all adjustable weights of NN (i.e. all w_{ij} and b_i. Generalizes to multiple outputs $\boldsymbol{y}$. Superscript n labels entries from the data set of size N.
- GD update: $\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \, \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{Y})$.

Calculating gradients: Back-propagation Part 2

- Select w_{ij} in arbitrary layer that connects from the unit j to unit i and calculate $\frac{\partial}{\partial w_{ij}} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w}))$ for all w_{ij}.
- Rewrite this as follows:

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w})) &= \frac{\partial}{\partial s_i} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w})) \frac{\partial s_i}{\partial w_{ij}} \\ &= \frac{\partial}{\partial s_i} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w})) a_j.\end{aligned}$$

- Define δ-error at unit i as $\delta_i := \frac{\partial}{\partial s_i} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w}))$.
- Yields: $\frac{\partial}{\partial w_{ij}} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w})) = \delta_i \, a_j$.

Calculating gradients: Back-propagation Part 3: Delta errors at the output layers.

- The δ-error at the output units, denoted as δ_k with preactivation s_k and activation $a_k = g(\boldsymbol{x}^n; \boldsymbol{w})$ for $Q - K + 1 \leq k \leq Q$ is $\delta_k = \frac{\partial}{\partial a_k} L(\boldsymbol{y}^n, g(\boldsymbol{x}^n; \boldsymbol{w}))\, f'(s_k)$
- f': derivative of activation function f of output layer.
- f in output layer is usually different than in hidden layers.
- Expression typically results in a difference between network outputs, i.e. predicted labels, and true labels.

Loss function		output activation	delta-error
squared loss	$1/2(a-y)^2$	linear: $f(x) = x$	$\delta = a - y$
binary CE	$-y\log(a) - (1-y)\log(1-a)$	sigmoid: $f(x) = \frac{1}{1+e^{-x}}$	$\delta = a - y$
categorical CE	$-\sum_{k=1}^{K} y_k \log(a_k)$	softmax: $f(\boldsymbol{x}) = \mathrm{softmax}(\boldsymbol{x})$	$\boldsymbol{\delta} = \boldsymbol{a} - \boldsymbol{y}$

Calculating gradients: Back-propagation Part 4: Delta errors at hidden layers.

- The δ-error at units not in the output layer, i.e. hidden layers, reads:

$$\delta_j = \frac{\partial}{\partial s_j} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w})) = \sum_i \frac{\partial}{\partial s_i} L(\boldsymbol{y}^n, \boldsymbol{g}(\boldsymbol{x}^n; \boldsymbol{w})) \frac{\partial s_i}{\partial s_j}$$

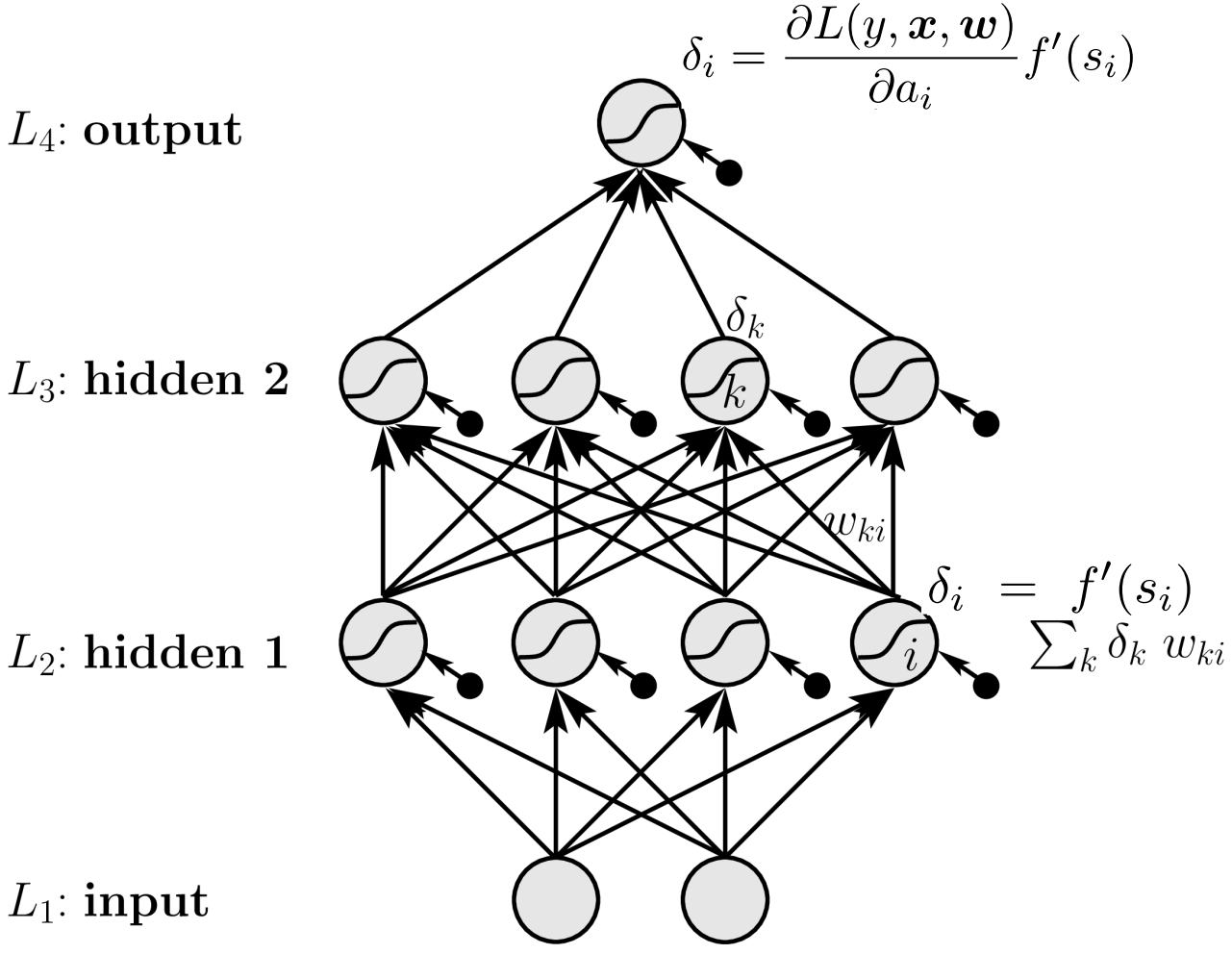
$$= f'(s_j) \sum_i \delta_i \, w_{ij},$$

- This is a recursive formula (lower layer unit error as function of above layer unit errors), allowing us to calculate all δ-errors of the network.
- $\sum_i$ goes over all i for which w_{ij} exists.
- Typically, i goes over all units in the layer *above* the layer where unit j is located.

Calculating gradients: Back-propagation Part 5: Recap of the algorithm.

- Select a sample x randomly from the training set and perform a forward-pass. Memorize activations and preactivations of all neurons.
- Calculate delta-errors at output units according to Part 3.
- Calculate delta-errors for hidden layers starting from the hidden layers closest to the output layer, thus backpropagating the error signal according to Part 4.
- Calculate gradients for weights as done in Part 2.
- Update the weights by performing a gradient descent step using the weight changes: $w_{ij}^{\text{new}} = w_{ij}^{\text{old}} - \eta\, \delta_i\, a_j$. The learning rate η must be chosen appropriately for the algorithm to converge. (Typical learning rates are $\eta = 0.1$ or $\eta = 0.01$).

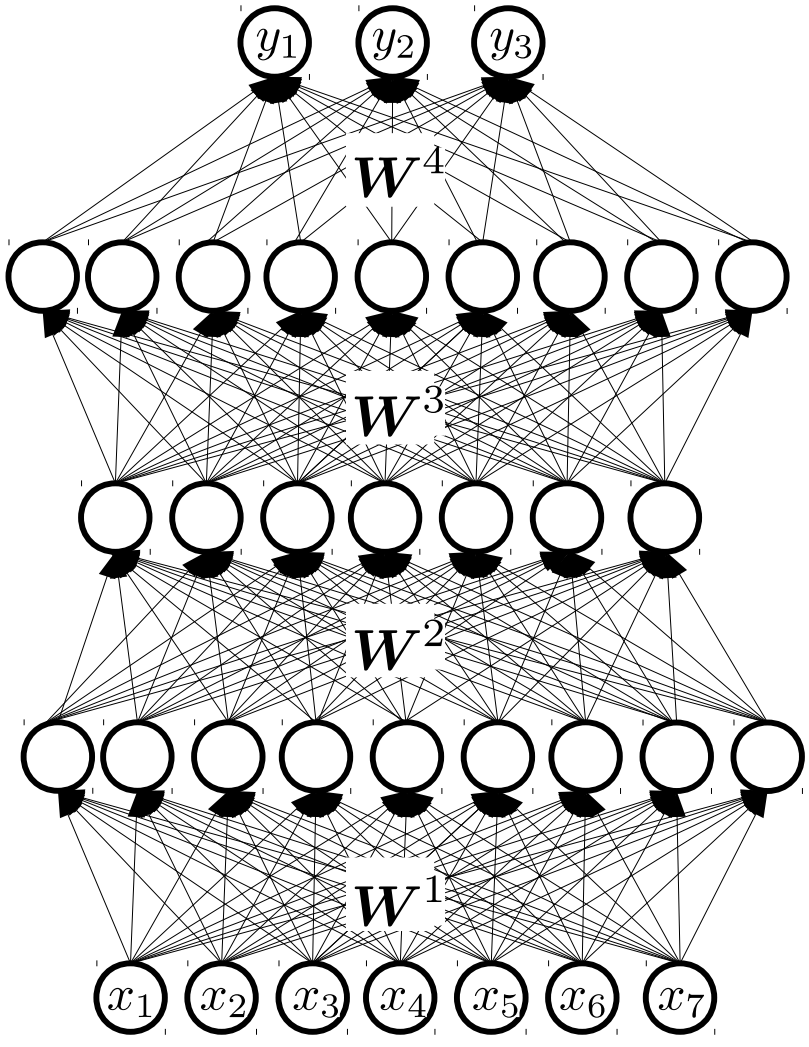
Calculating gradients: Back-propagation Part 6: Visualization of 4-layer NN.



Nowadays used notation for NNs: Matrix-vector notation

- First introduce the following expressions:
  - $\boldsymbol{x}$: input for layer 0; equivalent to $\boldsymbol{a}^{[0]}$
  - $\boldsymbol{W}^{[l]}$: weight matrix connecting layer $l-1$ and layer l
  - $\boldsymbol{s}^{[l]}$: pre-activations of layer l
  - $\boldsymbol{a}^{[l]}$: activations of layer l
  - f: activation function applied element-wise to vector.
- Given inputs $\boldsymbol{x}^{[l]}$ to a layer l, i.e. the activations $\boldsymbol{a}^{[l-1]}$ of the lower layer $l-1$, a NN computes activations of next layer as follows: $\boldsymbol{a}^{[l]} = f(\boldsymbol{s}^{[l]}) = f(\boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]})$
- Function from the input layer $\boldsymbol{x}$ to output layer: $\hat{\boldsymbol{y}} = g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]}) = \sigma(\boldsymbol{W}^{[L]}(\ldots f(\boldsymbol{W}^{[2]} f(\boldsymbol{W}^{[1]}\boldsymbol{x}))\ldots),$
- $\hat{\boldsymbol{y}}$ are activations of output layer and $\boldsymbol{x}$ is input of network. Outermost activation function σ typically different from activation function f in hidden layers.

Schematic view of a deep feed-forward neural network in matrix-vector notation



Backpropagation in Matrix-vector notation

- Now we can write:

$$\frac{\partial}{\partial w_{ij}^{[1]}} L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]})) = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}} \cdots \frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}} \cdots \frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}.$$

- Introduce delta errors as follows: $\boldsymbol{\delta}^{[l]} = \left(\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{[l]}} \right)^T$.
- Note that we typically use these deltas ${\boldsymbol{\delta}^{[l]}}^T$ as row vectors.
- Similar arguments as before lead to recursive formula for delta errors:

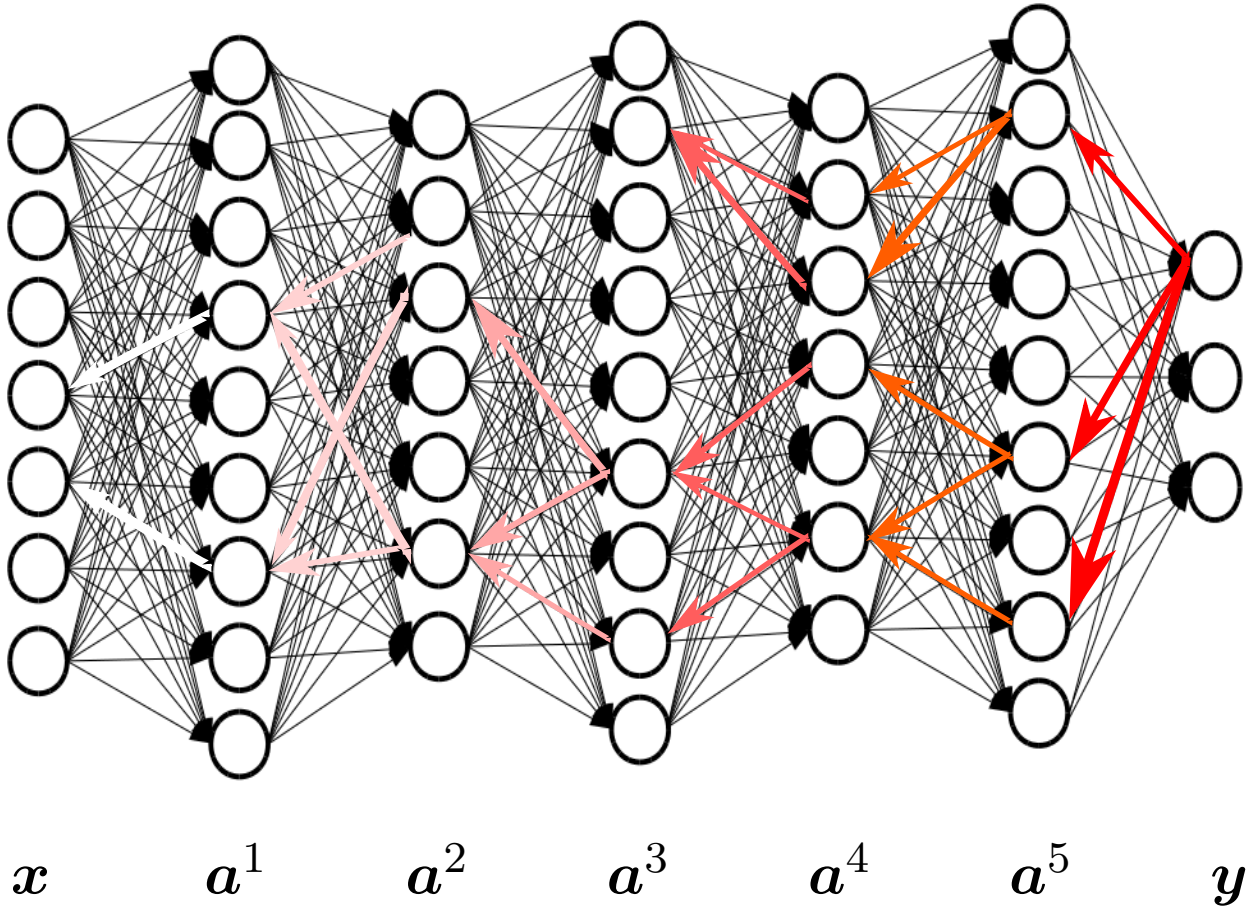
$${\boldsymbol{\delta}^{[l-1]}}^T = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{[l]}} \frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}} = {\boldsymbol{\delta}^{[l]}}^T \underbrace{\boldsymbol{W}^{[l]} \operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]}) \right)}_{\boldsymbol{J}^{[l]}}.$$

Vanishing and exploding gradients

- $\boldsymbol{\delta}^{[l-1]^T} = \boldsymbol{\delta}^{[l]^T} \boldsymbol{W}^{[l]} \operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]})\right)$:
- Important property that influences the ability of a NN to learn: size (norm) of the delta-errors that backpropagate through the network.
- Since for each layer delta errors are multiplied by Jacobian $\rightarrow$ exponential behavior (growth or shrinkage):
  - $\|\boldsymbol{\delta}^{[l-1]}\| < \|\boldsymbol{\delta}^{[l]}\|$: Vanishing gradients: Has been typical case since the derivatives of the sigmoid activation function is at most $|f'(x)| = 0.25 \rightarrow \|\operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]})\right)\| \leq 0.25 \rightarrow$ norm of delta-errors becomes smaller through each layer (if not compensated by the norm of $\boldsymbol{W}^{[l]}$).
  - $\|\boldsymbol{\delta}^{[l-1]}\| \approx \|\boldsymbol{\delta}^{[l]}\|$: Stable gradients: Ideal case that the delta errors have a similar norm in each layer; many recent algorithmic improvements aim at keeping this quantity close to one (e.g. initialization strategies).
  - $\|\boldsymbol{\delta}^{[l-1]}\| > \|\boldsymbol{\delta}^{[l]}\|$: Exploding gradients: If norm of weight matrix is large $\rightarrow$ norm of delta-errors grows through each layer $\rightarrow$ unstable learning or even numeric overflows.

Vanishing Gradient: Visualization

- Gradient signal gets lost in the noise:



Example: ReLUs and ELUs

- **Re**ctified **l**inear **u**nit, by Nair and Hinton in 2010:
  - Idea: N sigmoids with shared weights but different biases:

$$\sum_{i}^{N} \sigma(h - i + 0.5) \approx \log(1 + e^{h}) \approx max(0, h)$$

    where $h = \mathbf{w} \cdot x + b$
  - Gradient is either 0 or 1 (helps against vanishing gradients!)
  - ReLU nets learn a piecewise linear function
  - Problem: Dying ReLUs
- **E**xponential **l**inear **u**nit, by Clevert, Unterthiner, Hochreiter in 2015: