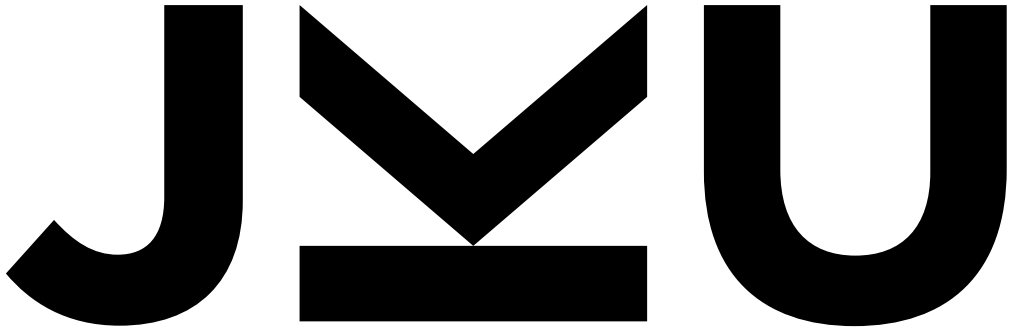
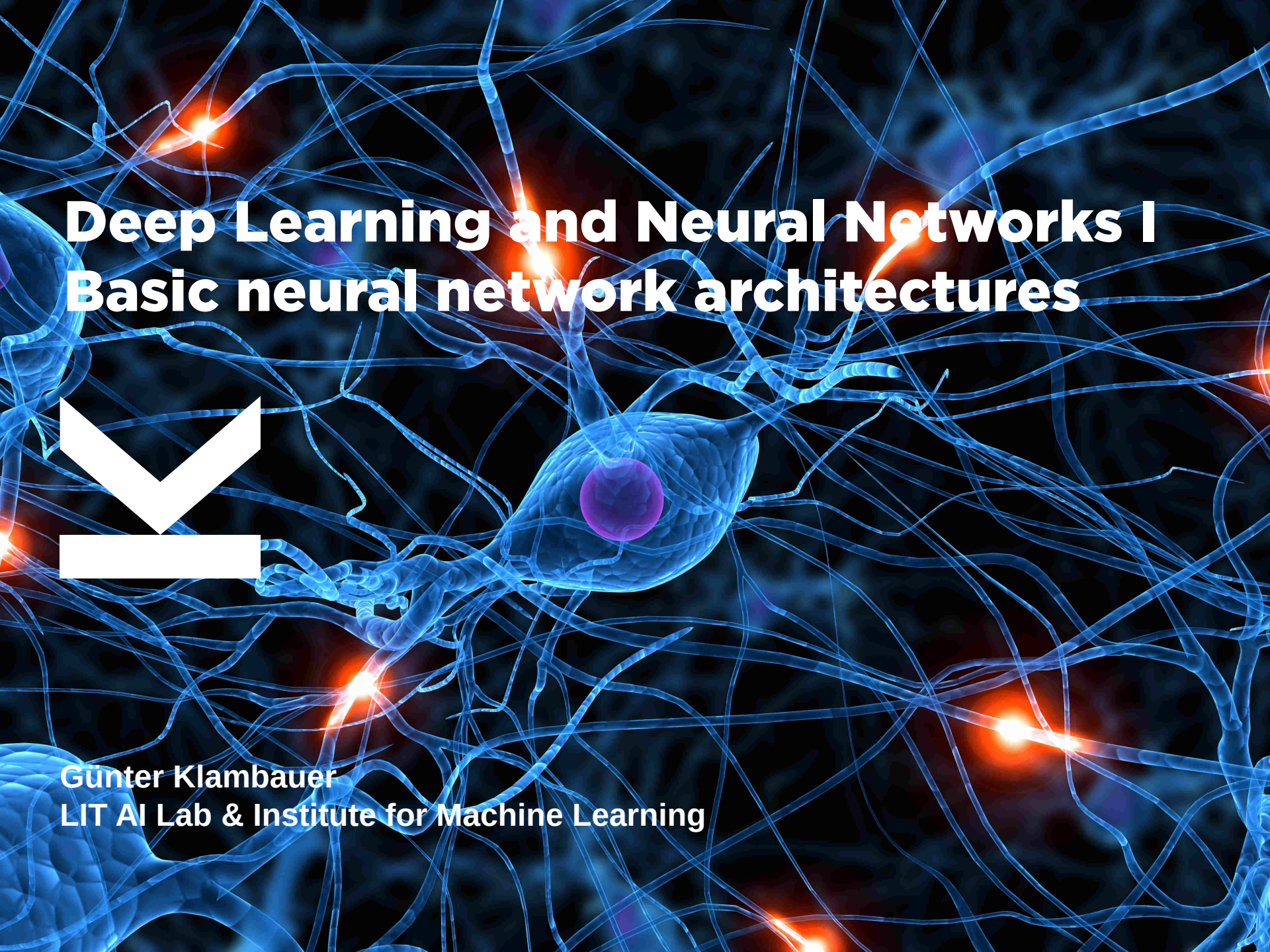


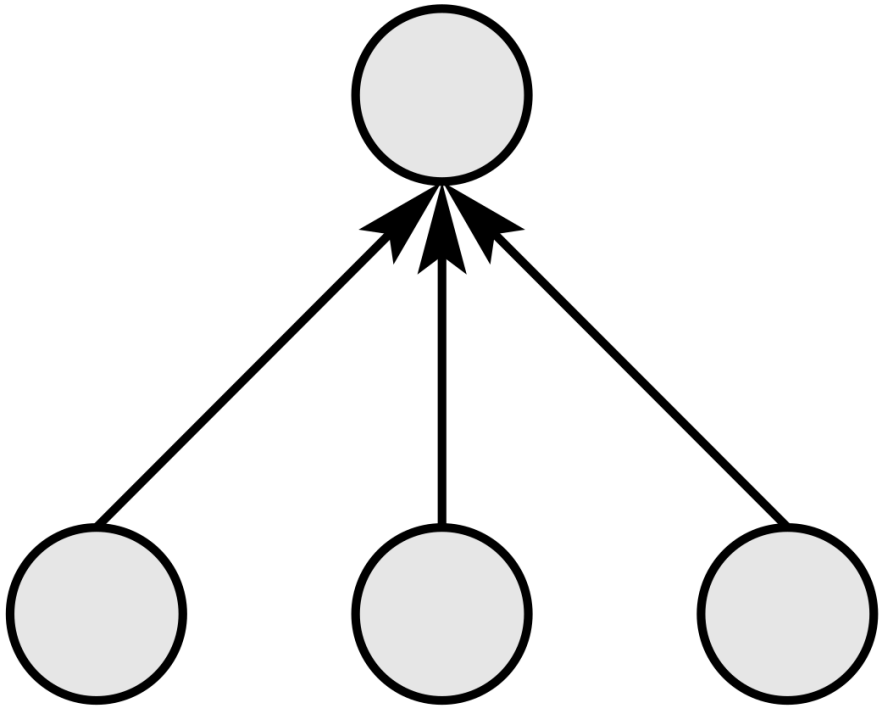
JOHANNES KEPLER
UNIVERSITY LINZ



Deep Learning and Neural Networks I
Basic neural network architectures

Günter Klambauer
LIT AI Lab & Institute for Machine Learning

Artificial neural networks: linear neuron



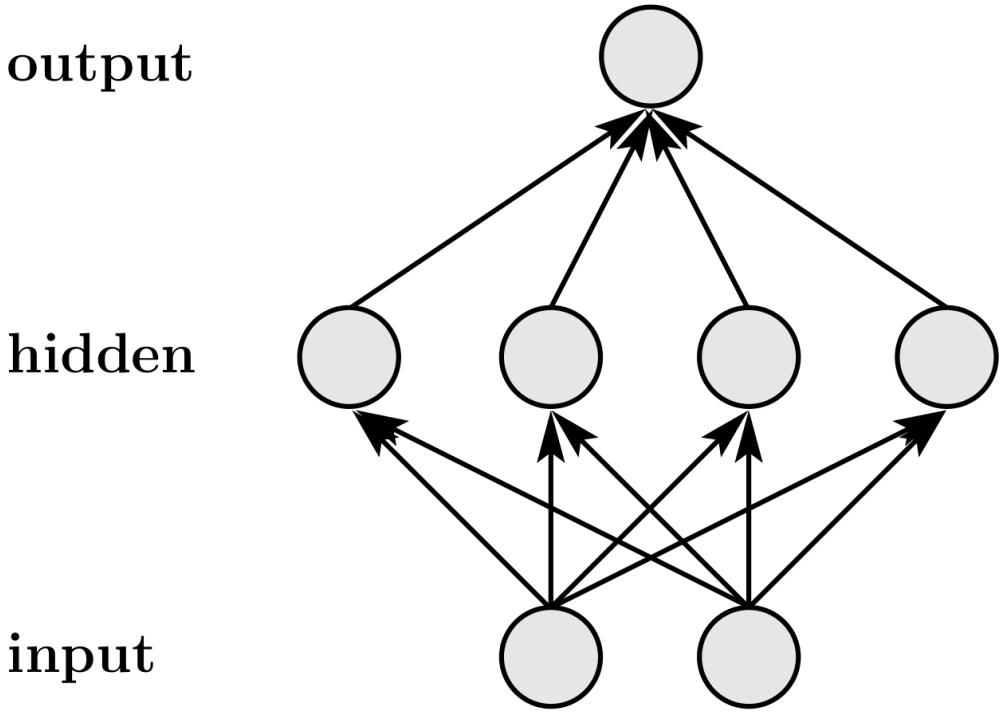
$$a = \boldsymbol{w}^T \boldsymbol{x}$$

$\boldsymbol{w}$

$\boldsymbol{x}$

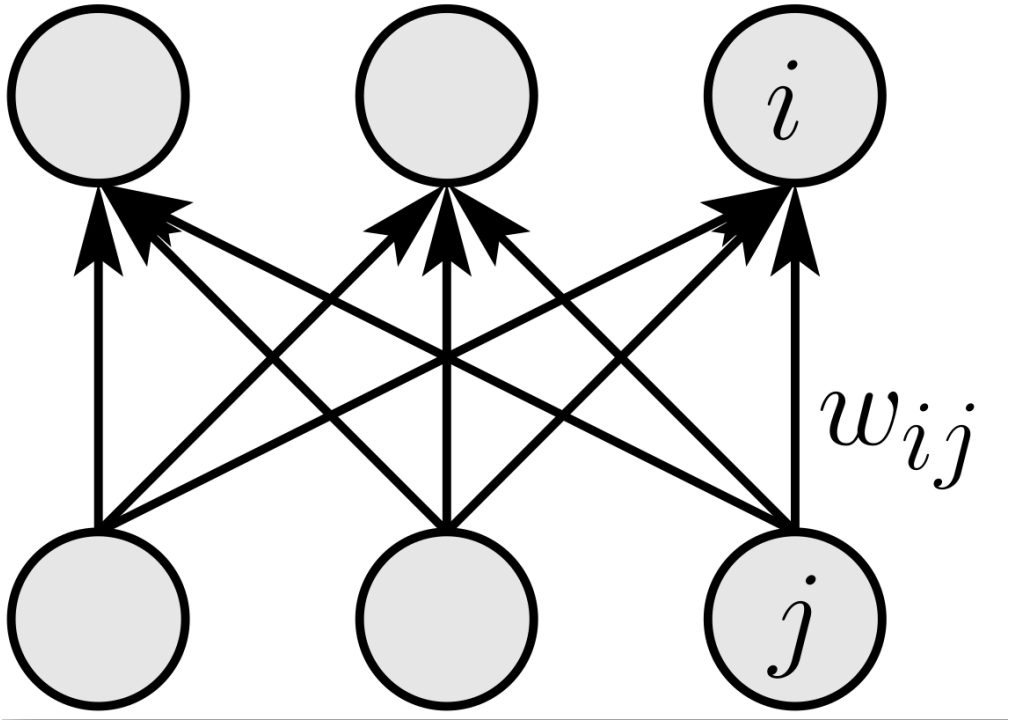
- Inputs directly connected to output
- Input: $\boldsymbol{x}$ adaptive weights: $\boldsymbol{w}$ output (activation): a

Artifical neural networks: 3-layer network



- Artificial neural networks: a 3-layered net with an input, hidden, and output layer

Neural network with adaptive weights



Artifical neural networks: units and weights. The weight w_{ij} gives the weight, connection strength, or synaptic weight from unit j to unit i

Overview

- 4. Basic neural network architectures
  - 4.5 Single-layer networks cannot solve XOR
  - 4.6 Multi-layer perceptron
    - 4.6.1. Model and forward pass of an MLP
    - 4.6.2. Gradient descent and the backpropagation algorithm
    - 4.6.3. Training MLPs
  - 4.7. (Deep) feed-forward neural networks
    - Backpropagation in DNNs

A quick task for you

-0.02, 0.01 → label: 0

1.05, 0.99 → label: 0

0.03, 0.01 → label: 0

0.97, -0.01 → label: 1

0.02, 0.98 → label: 1

1.03, -0.02 → label: 1

1.03, 1.01 → label: 0

0.96, 0.03 → label: 1

1.00, 0.01 → label: ?

-0.04, 0.03 → label: ?

4.5 Single-layer networks cannot solve XOR

- Assume we have

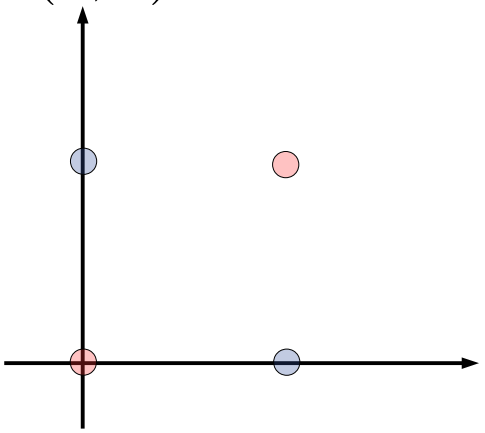
  $\boldsymbol{x}_1 = (0, 0),\ \boldsymbol{x}_2 = (1, 0),\ \boldsymbol{x}_3 = (0, 1),\ \text{and}\ \boldsymbol{x}_4 = (1, 1)$

- With the labels

  $y_1 = 0, y_2 = 1,\ y_3 = 1,\ \text{and}\ y_4 = 0$

- And a linear (single-layer) network

$$g_1(\boldsymbol{x}; \boldsymbol{w}) = x_1 w_1 + x_2 w_2$$



- There are no parameters that solve this problem

4.5 Single-layer networks cannot solve XOR

- No parameters solve this problem

$$
\begin{aligned}
0 &= 0\ w_1 + 0\ w_2 \\
1 &= 1\ w_1 + 0\ w_2 \\
1 &= 0\ w_1 + 1\ w_2 \\
0 &= 1\ w_1 + 1\ w_2.
\end{aligned}
$$

- Adding sigmoids or softmax activations does not help
- Solutions?

4.5 Single-layer networks cannot solve XOR

- Use a hidden layer or two-layer network

$$g_2(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{w}, \boldsymbol{b}) = \boldsymbol{w}^T \max(0, \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{w}^T \boldsymbol{h}$$

- We can find parameters that solve the problem. For example:

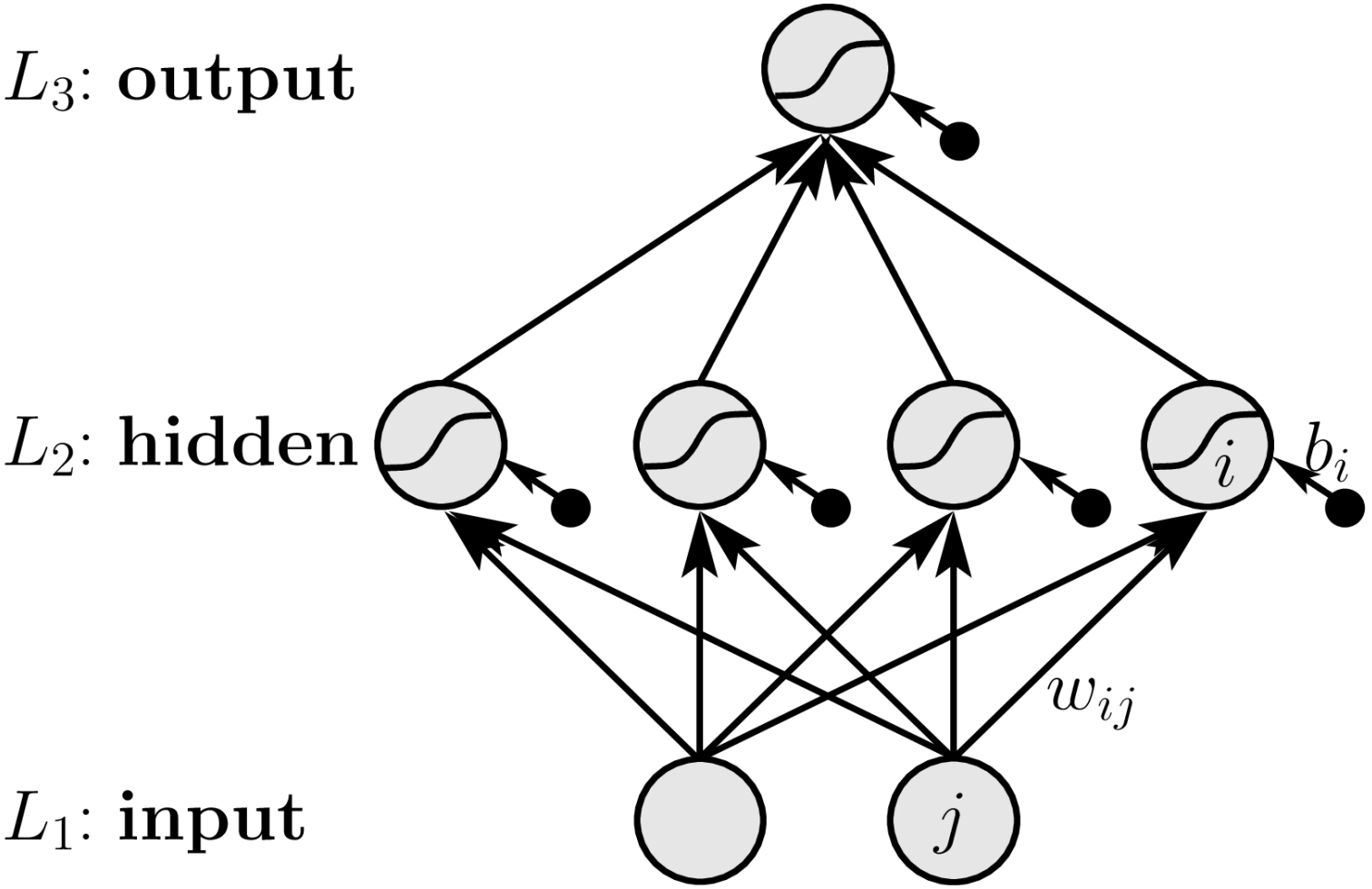
$$\boldsymbol{W} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \text{ and } \boldsymbol{w} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}$$

- The transformation $\max(0, \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ has mapped the points into a space where they are linearly separable

Summary on single-layer networks and XOR

- We have indicated that single-layer networks cannot solve XOR
- A simple two-layer network has solved it (using *ReLU non-linearity*)
- Multi-layer perceptrons are even universal function approximators
  - → Universal function approximator theorem (Hornik, 1989)

Multi-layer perceptron



Multi-layer perceptron: quantities

- a_i: *activity* of the i-th unit

Multi-layer perceptron: quantities

- s_i: *network input* (pre-activation) to the i-th unit ($i > D$) computed as

$$s_i = \sum_{j=0}^{Q} w_{ij} a_j \tag{1}$$

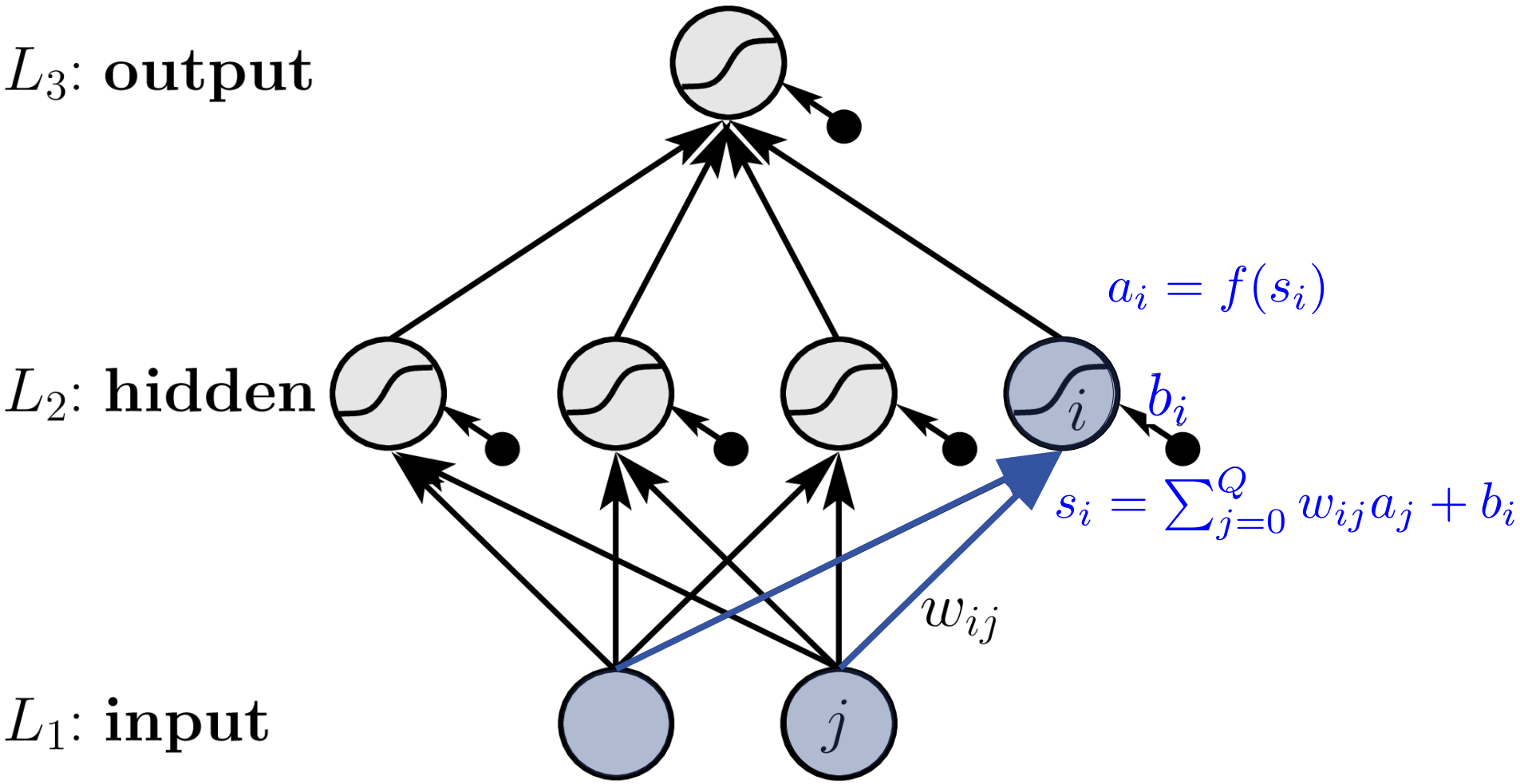
- f: *activation function* with

$$a_i = f(s_i) \tag{2}$$

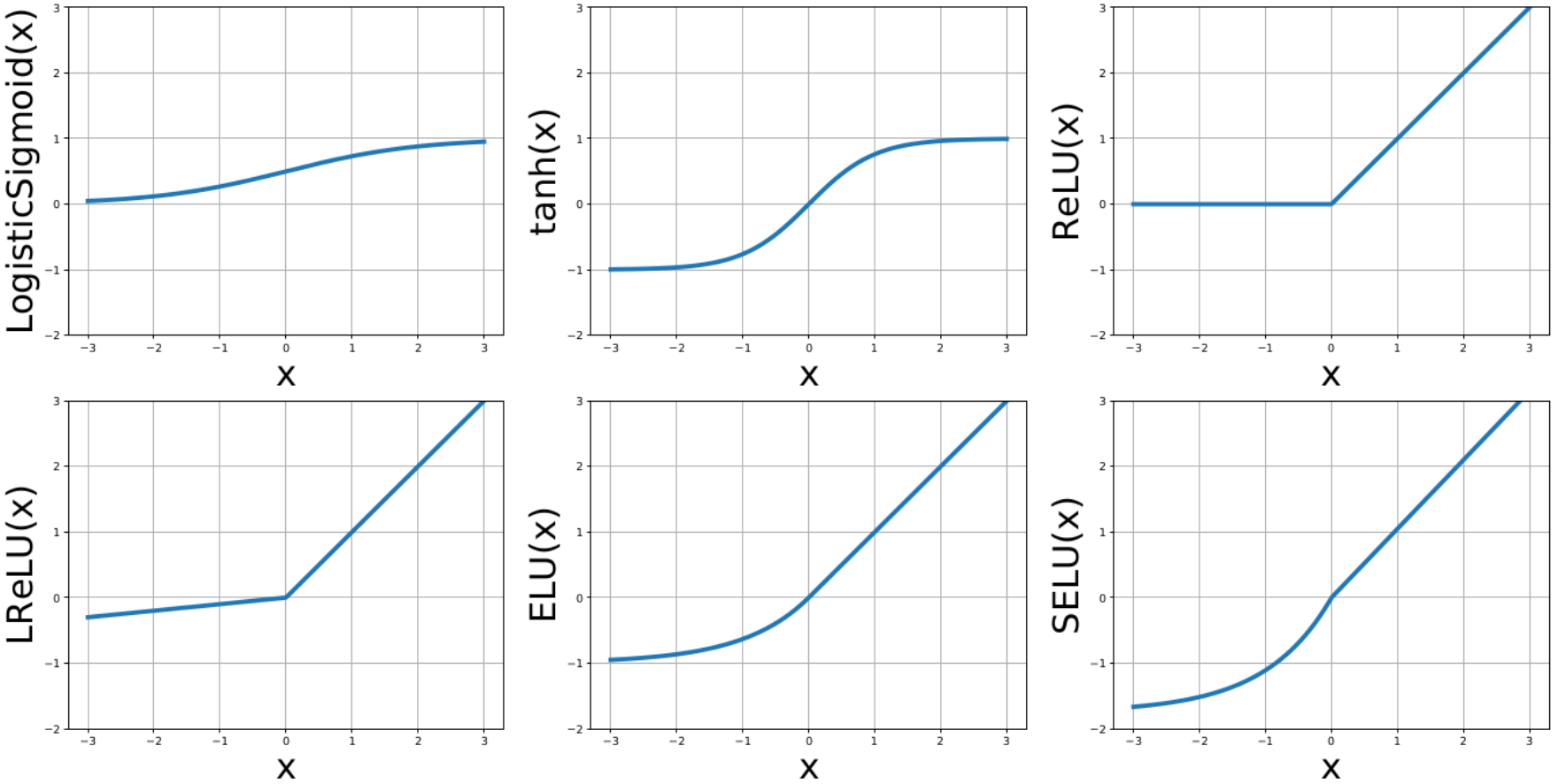
  It is possible to define different activation functions f_i for different units. The activation function is sometimes called *transfer function* (in more realistic networks one can distinguish between activation of a neuron and the signal which is transferred to other neurons).

- the *architecture* of a neural network is given through number of layers, units in the layers, and defined connections between units – the activations function may be accounted to the architecture.

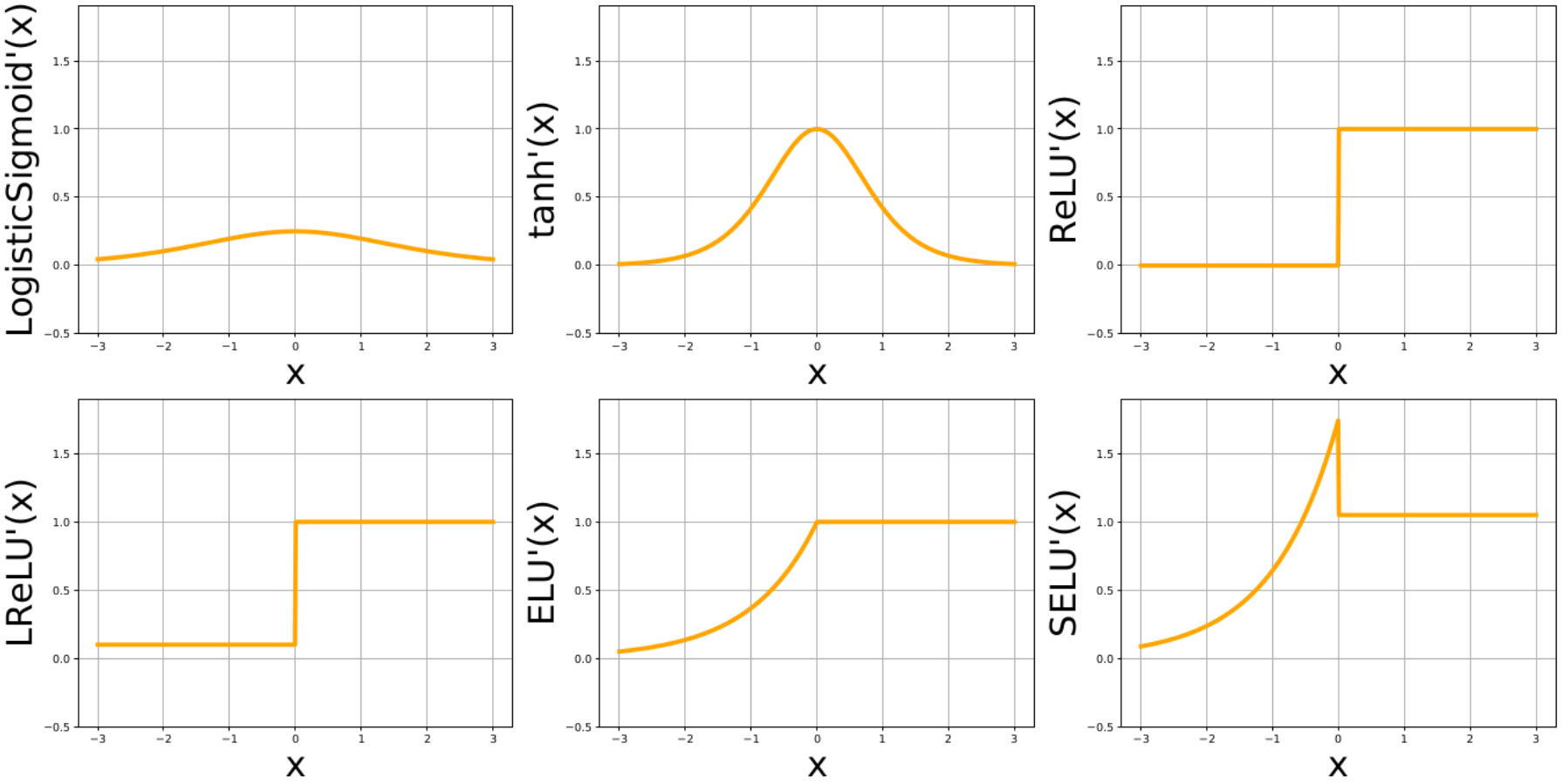
Multi-layer perceptron



A note on activation functions

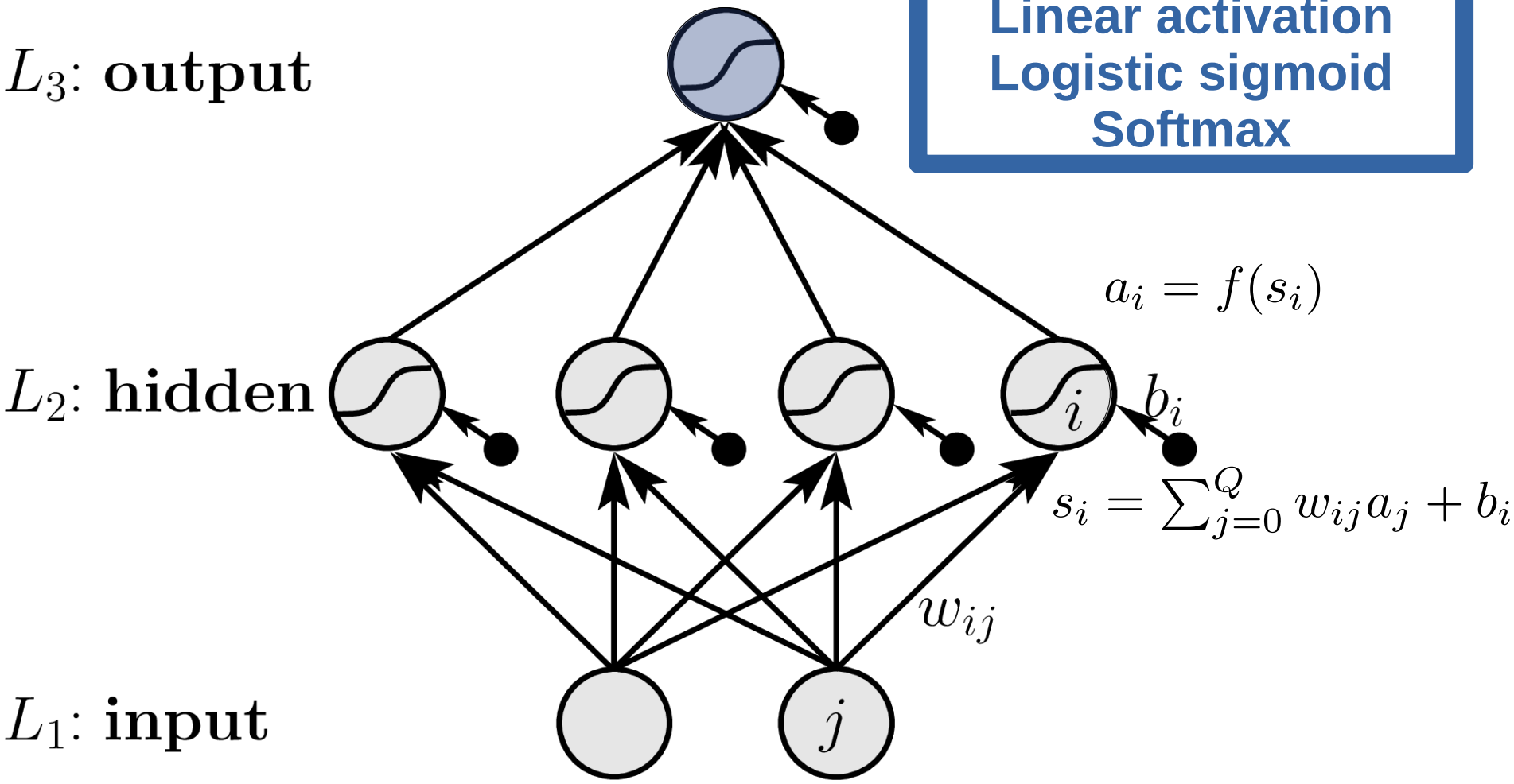


A note on activation functions: derivatives



Multi-layer perceptron: activations at output layer

Usually:
Linear activation
Logistic sigmoid
Softmax



Forward-pass of an MLP: pseudo-code

Algorithm .1 Forward Pass of an MLP

BEGIN initialization
  provide input $\boldsymbol{x}$
  for all $(i = 1; i \leq D;\ i++)$ **do**
    $a_i = x_i$
  end for
END initialization
BEGIN Forward Pass
  for $(\nu = 2; \nu \leq L; \nu++)$ **do**
    for all $i \in L_\nu$ **do**

$$s_i = \sum_{j=0; w_{ij} \text{ exists}}^{Q} w_{ij} a_j$$

      $a_i = f(s_i)$
    end for
  end for
  provide output $\hat{y}_i = (g(\boldsymbol{x}; \boldsymbol{w}))_i = a_i$, for all $Q - K + 1 \leq i \leq Q$
END Forward Pass

4.6.2. Gradient descent and the backpropagation algorithm

- Training of MLPs by *back-propagation*
- Famous algorithm made popular by Rumelhart et al. (1986); proposed earlier by Werbos (1974).
- Also called *delta-propagation*

Backpropagation

- Same setting as before: we aim at minimizing the empirical error by gradient descen

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

  - $g(\boldsymbol{x}^n, \boldsymbol{w})$ is the network and $\boldsymbol{w}$ contains all weights (weight matrices) in all layers and (potentially) biases

- We want to perform a gradient descent update

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \, \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{Y}, \boldsymbol{X}, \boldsymbol{w})$$

  or for a single data point:

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \, \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{x}, \boldsymbol{w})$$

Backpropagation: derivative w.r.t. weights

- We derive the loss w.r.t. the weights:

$$\frac{\partial}{\partial w_{ij}} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right)$$

- Using the chain rule, we obtain:

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

where we used $\frac{\partial s_i}{\partial w_{ij}} = a_j$

Backpropagation: Defining delta errors

- We define the delta error at unit i as:

$$\delta_i := \frac{\partial}{\partial s_i} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right)$$

- And we obtain:

$$\frac{\partial}{\partial w_{ij}} L\left(\boldsymbol{y}^n, \boldsymbol{g}\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) = \delta_i a_j.$$

- Compare a result from before (single-layer nets):

$$\frac{\partial}{\partial w_{ij}} R_{\text{emp}}(\boldsymbol{w}, \boldsymbol{x}, y) = \underbrace{\left(\text{softmax}(\boldsymbol{W}\boldsymbol{x})_i - y_i\right)}_{=\delta_i} \underbrace{x_j}_{=a_j},$$

Backpropagation: delta error at output units

- Delta errors at output units:

$$\delta_k = \frac{\partial}{\partial a_k} L\left(\boldsymbol{y}^n, g\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) f'(s_k)$$

  where we have

  - activations $a_k = g\left(\boldsymbol{x}^n; \boldsymbol{w}\right)$ $\quad Q - K + 1 \leq k \leq Q$
  - pre-activations s_k $\quad Q - K + 1 \leq k \leq Q$
    at the output layer

- This is the same as for linear, log., softmax regr

Backpropagation: delta error at hidden units

- The delta errors at the hidden units are

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

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

- where the sum goes over all units for which the weight exists (the units in the layer above)

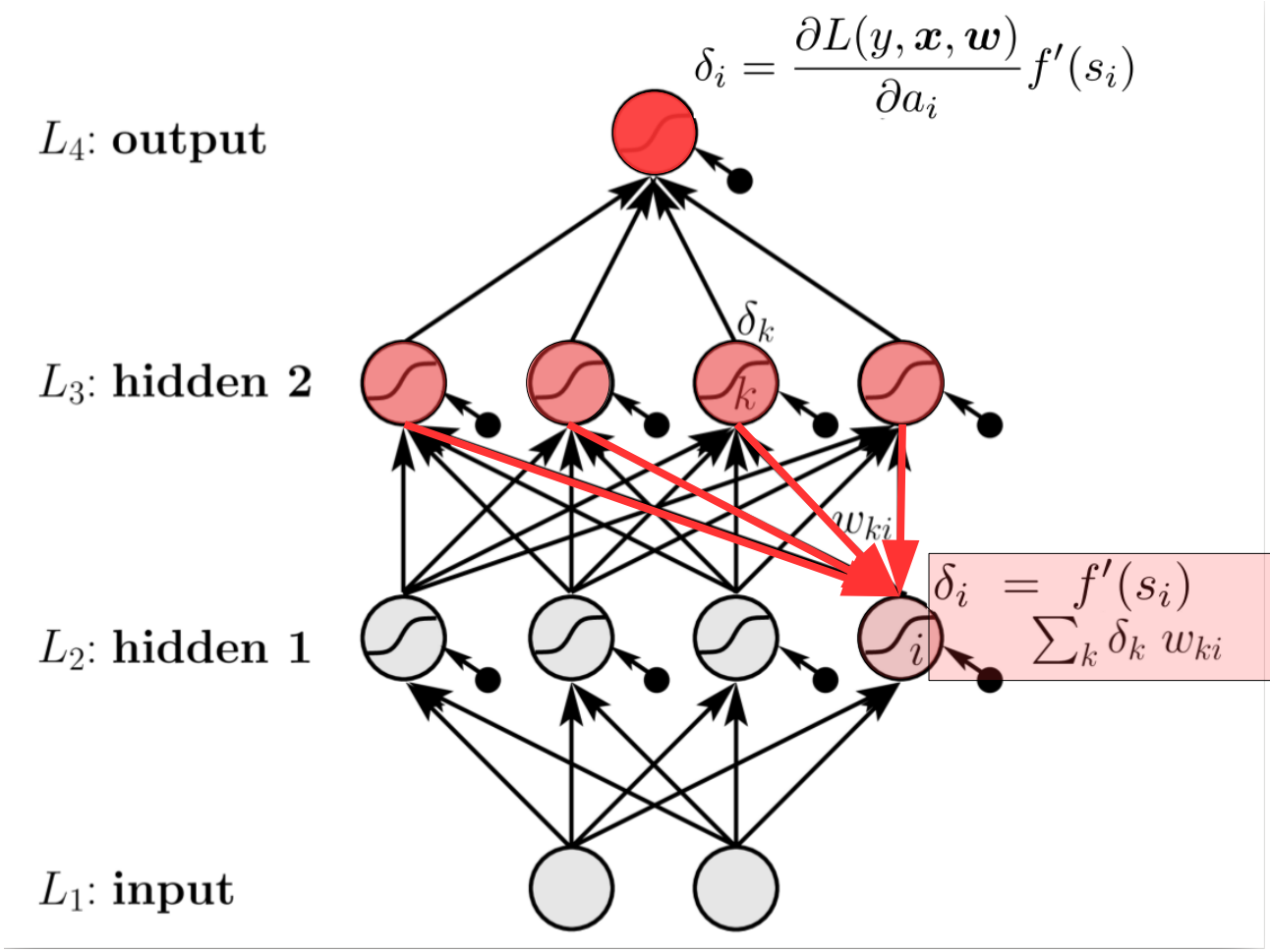
Loss functions and their corresponding output activation functions

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: $f(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x})$	$\boldsymbol{\delta} = (\boldsymbol{a}-\boldsymbol{y})$

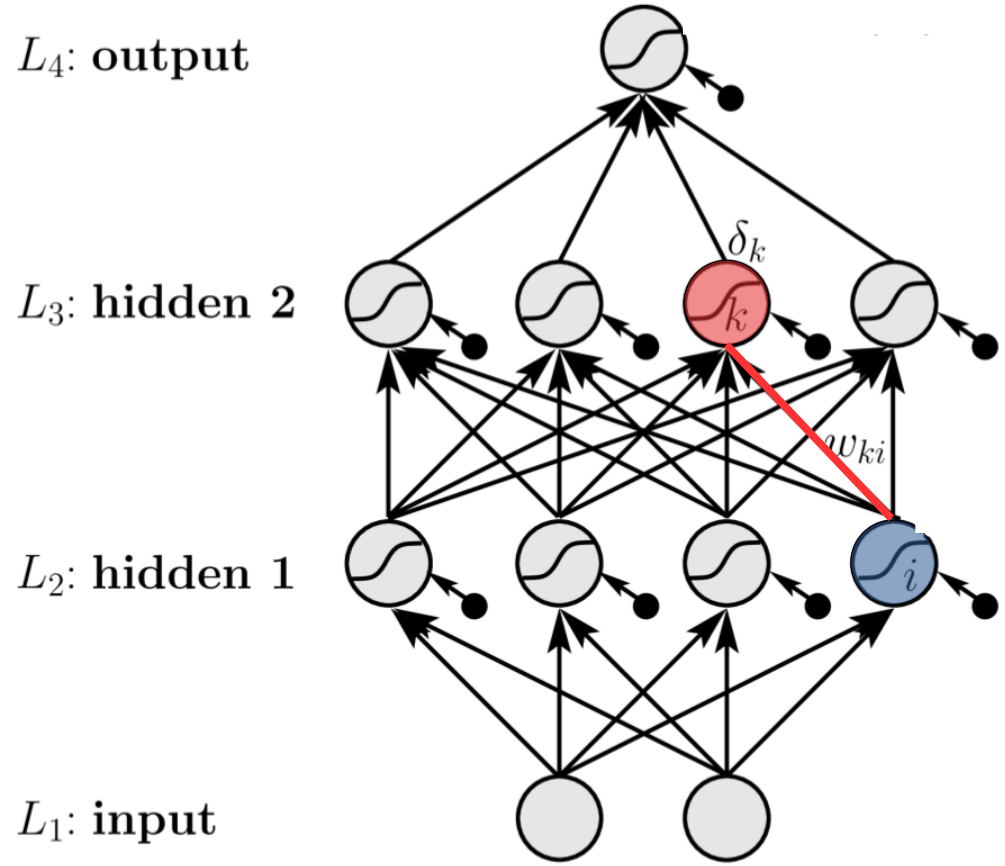
Table 4.1: Frequently used loss functions for neural networks and their corresponding activation functions at the output units.

- Potential problems of other combinations?

Delta propagation



Weight update



$$\frac{\partial L}{\partial w_{ki}} = \delta_k a_i$$

Recap: backpropagation

- 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: $\delta_k = \frac{\partial}{\partial a_k} L\left(\boldsymbol{y}^n, g\left(\boldsymbol{x}^n; \boldsymbol{w}\right)\right) f'(s_k)$
- Calculate delta-errors for hidden layers starting from the hidden layers closest to the output layer, thus backpropagating the error signal according to $\delta_j = f'(s_j) \sum_i \delta_i w_{ij}$
- Calculate gradients for weights according to $\frac{\partial L}{\partial w_{ij}} = \delta_i a_j$
- Update the weights by performing a gradient descent step using the weight changes $\Delta w_{ij} = -\eta \delta_i \; a_j$

Backpropagation: pseudo-code

Algorithm .1 Backward Pass of an MLP

BEGIN initialization
  provide activations a_i of the forward pass and the label y
  for $(i = Q - K + 1; i \leq Q; i++)$ **do**
    $\delta_i = \dfrac{\partial L(y, \boldsymbol{x}, \boldsymbol{w})}{\partial a_i} f'(s_i)$
    for all $j \in L_{L-1}$ **do**
      $\Delta w_{ij} = -\eta \delta_i a_j$
    end for
  end for
END initialization
BEGIN Backward Pass
  for $(\nu = L - 1; \nu \geq 2; \nu --)$ **do**
    for all $i \in L_\nu$ **do**
      $\delta_i = f'(s_i) \sum_k \delta_k w_{ki}$
      for all $j \in L_{\nu-1}$ **do**
        $\Delta w_{ij} = -\eta \delta_i a_j$
      end for
    end for
  end for
END Backward Pass

Remark: efficiency of backprop

- The name “backprop” refers to the fact that errors (deltas) because the delta errors of one layer are used to calculate the deltas for the layer below
- Complexity is $\mathcal{O}(W)$

- A similar approach can be used to calculate the second derivative (not detailed now)

Remark: full batch, stochastic and online learning

- The gradient can be calculated on the full training set, and then an update step can be taken: *full-batch gradient*
- Calculate gradient on single sample and perform update: *online learning*
- Sample a small subset, e.g. 32, samples, calc gradient and perform update: *minibatch training or stochastic gradient desc.*

Training MLPs: architecture and hyperparams

- MLPs now offer many choices
  - Number of layers
  - Number of neurons
  - Activation function
  - Learning rate
- Minimizing the empirical error is not sufficient (see Chapter 5)
  - Optimize architecture on *validation set*
  - Evaluate final model on *test set*

Questions and left-overs from last lecture

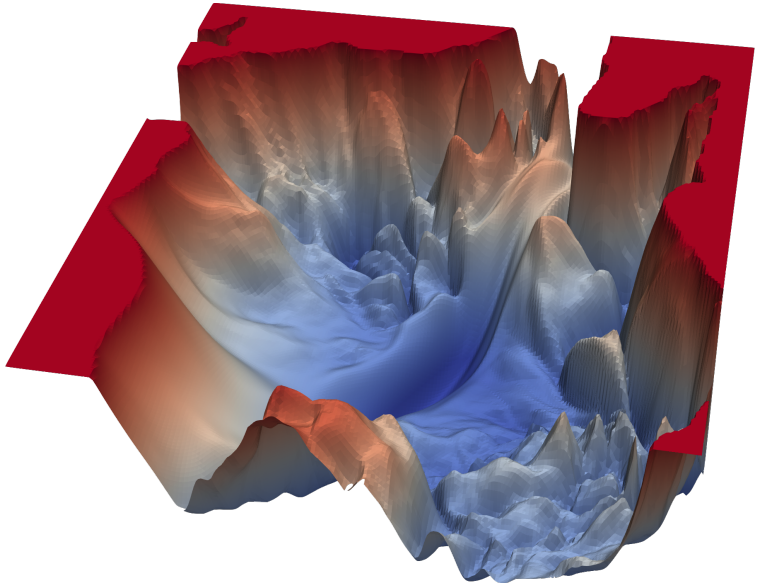
- Gradient of softmax regression: done in *Exercises*
- Typo on notation slide: see next slide

Overview

- 4. Basic neural network architectures
  - 4.5 Single-layer networks cannot solve XOR
  - 4.6 Multi-layer perceptron
  - 4.7 (Deep) feed-forward neural networks
    - Backpropagation in DNNs
  - 4.8 Jacobian
  - 4.9 Hessian
  - 4.10 Tricks of the trade

Remark: optimization problem and loss surface

- Loss surface is highly non-convex
- Uniqueness? No! *Weight space symmetries*
  - If there is a parameter set with horizontal tangent, we can easily switch signs to get an equivalent set of parameters (e.g. tanh)
  - Also the order of neurons can be changed
- However: Some theoretic results that optimization by SGD still yields ‘good’ minima (see Part II)



Li, H., Xu, Z., Taylor, G., Studer, C., & Goldstein, T. (2018). Visualizing the loss landscape of neural nets. In Advances in Neural Information Processing Systems (pp. 6389-6399).

MLPs: Summary

- XOR problems
- Architecture of MLPs
  - Forward pass
  - Backward pass (backpropagation)
- Remarks on optimization, etc

Deep feed-forward neural networks

- Same principle as MLPs
  - Layers of interconnected neurons
  - More layers; more neurons
  - Different activation functions

Table 1: Comparison of typical multi-layer perceptrons and deep feed-forward neural networks (informal guidelines).

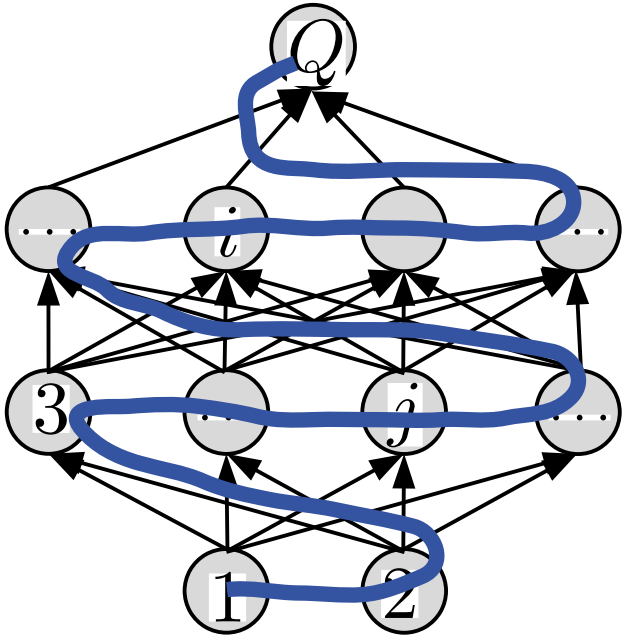
	MLP	**DNN**
Number of hidden layers	1 or few	> 1 up to several hundreds
Number of neurons per layer	< 10	> 100
Activation function	tanh, sigmoid	ReLU, SELU

DNNs: notation

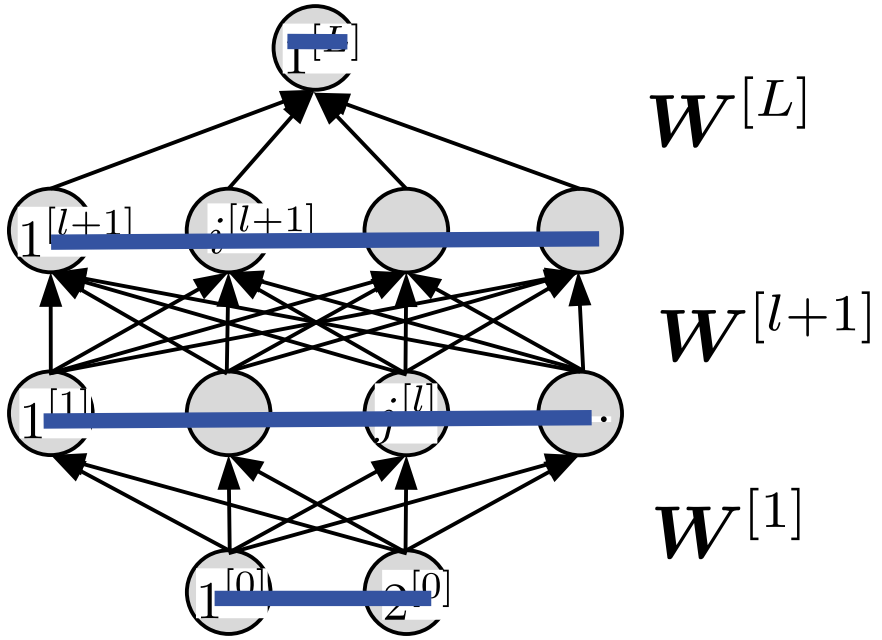
- $\boldsymbol{x}$: input for layer 0; equivalent to $\boldsymbol{a}^{[0]}$
- $\boldsymbol{W}^{[l]}$: weight matrix connecting layer $l-1$ and layer l
- $\boldsymbol{s}^{[l]}$: pre-activations of layer l
- $\boldsymbol{a}^{[l]}$: activations of layer l
- f: activation function that is applied element-wise to a vector.

Remark: Notation for MLP and for deep FNNs

MLP:



FNN:



DNNs: Forward pass

- Activations from one layer to the next layer:

$$\boldsymbol{a}^{[l]} = f(\boldsymbol{s}^{[l]}) = f(\boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]})$$

- Full function:

$$\hat{\boldsymbol{y}} = g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]}) = \sigma(\boldsymbol{W}^{[L]}(\ldots f(\boldsymbol{W}^{[2]} f(\boldsymbol{W}^{[1]}\boldsymbol{x}))\ldots)$$

- With bias units

$$\boldsymbol{a}^{[l]} = f(\boldsymbol{s}^{[l]}) = f(\boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}).$$

Why non-linearities?

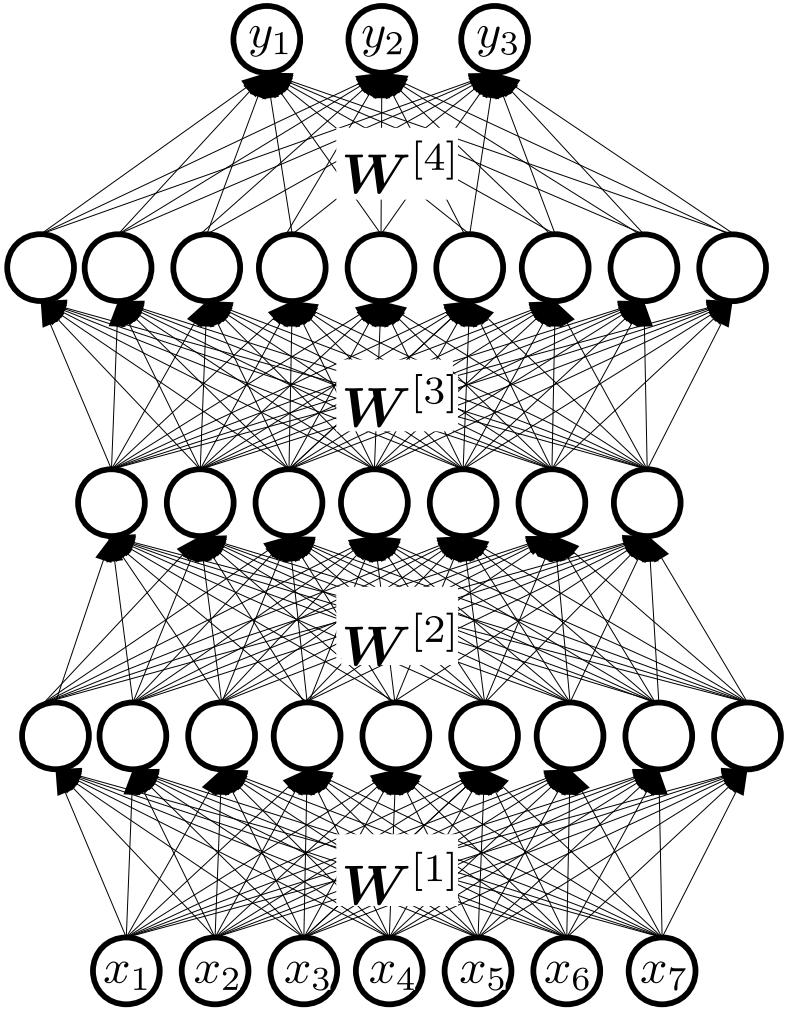
- With non-linearity:

$$g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]}) = \sigma(\boldsymbol{W}^{[L]}(\ldots f(\boldsymbol{W}^{[2]} f(\boldsymbol{W}^{[1]} \boldsymbol{x})) \ldots)$$

- Without non-linearity:

$$g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]}) = \sigma(\underbrace{\boldsymbol{W}^{[L]} \ldots \boldsymbol{W}^{[1]}}_{=\hat{\boldsymbol{W}}} \boldsymbol{x})$$

DNN: Forward pass



$$\hat{y} = \sigma(\underbrace{\boldsymbol{W}^{[4]}\ \boldsymbol{a}^{[3]}}_{=\boldsymbol{s}^{[4]}})$$

$$\boldsymbol{a}^{[3]} = f(\underbrace{\boldsymbol{W}^{[3]}\ \boldsymbol{a}^{[2]}}_{=\boldsymbol{s}^{[3]}})$$

$$\boldsymbol{a}^{[2]} = f(\underbrace{\boldsymbol{W}^{[2]}\ \boldsymbol{a}^{[1]}}_{=\boldsymbol{s}^{[2]}})$$

$$\boldsymbol{a}^{[1]} = f(\underbrace{\boldsymbol{W}^{[1]}\ \boldsymbol{x}}_{\boldsymbol{s}^{[1]}})$$

$$\boldsymbol{x}$$

Backpropagation in DNNs

- Without loss of generality, we derive with respect to a weight in the first layer:

$$\frac{\partial}{w_{ij}^{[1]}} L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]})) = \underbrace{\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}}_{=:A} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}}_{=:B} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}}_{=:C}$$

- Now we treat the expressions A, B and C.

Backpropagation in DNNs

- We first define:

$$\boldsymbol{\delta}^{[l]^T} := \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{[l]}}$$

Backpropagation in DNNs

- Without loss of generality, we derive with respect to a weight in the first layer:

$$\frac{\partial}{w_{ij}^{[1]}} L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]})) = \underbrace{\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}}_{=:A} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}}_{=:B} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}}_{=:C}$$

- Now we treat the expressions A, B and C.

Backpropagation in DNNs

- Without loss of generality, we derive with respect to a weight in the first layer:

$$\frac{\partial}{w_{ij}^{[1]}} L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]})) = \underbrace{\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}}_{=:A} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}}_{=:B} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}}_{=:C}$$

- A $\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}$: Depends on the choice of loss function and activation. If canonical links are chosen, this can result in

$$\boldsymbol{\delta}^{[L]^T} = (\hat{\boldsymbol{y}} - \boldsymbol{y})^T$$

Backpropagation in DNNs

- Without loss of generality, we derive with respect to a weight in the first layer:

$$\frac{\partial}{w_{ij}^{[1]}} L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]})) = \underbrace{\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}}_{=:A} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}}_{=:B} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}}_{=:C}$$

- B $\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}$:

$$\boldsymbol{s}^{[l]} = \boldsymbol{W}^{[l]} f(\boldsymbol{s}^{[l-1]})$$

$$\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}} = \boldsymbol{W}^{[l]} \operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]})\right)$$

Backpropagation in DNNs

- Without loss of generality, we derive with respect to a weight in the first layer:

$$\frac{\partial}{w_{ij}^{[1]}} L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]})) = \underbrace{\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}}_{=:A} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}}_{=:B} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}}_{=:C}$$

- *Recursive formula* for deltas:

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

Backpropagation in DNNs

- Without loss of generality, we derive with respect to a weight in the first layer:

$$\frac{\partial}{w_{ij}^{[1]}} L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L]})) = \underbrace{\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^{[L]}}}_{=:A} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[l]}}{\partial \boldsymbol{s}^{[l-1]}}}_{=:B} \cdot \ldots \cdot \underbrace{\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}}}_{=:C}$$

- C: $\frac{\partial \boldsymbol{s}^{[1]}}{\partial w_{ij}^{[1]}} = \frac{\partial}{\partial w_{ij}^{[1]}} \boldsymbol{W}^{[1]} \boldsymbol{a}^{[0]} = \begin{pmatrix} 0 \\ \vdots \\ a_j^{[0]} \\ \vdots \\ 0 \end{pmatrix}$,

  where the non-zero entry $a_j^{[0]}$ is at the i-th position.

Backpropagation in DNNs

- Overall we find

$$\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^{[l]}} = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{[l]}} \frac{\partial \boldsymbol{s}^{[l]}}{\partial w_{ij}^{[l]}} = (\boldsymbol{\delta}^{[l]})_i (\boldsymbol{a}^{[l-1]})_j$$

  or conveniently:

$$\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{W}^{[l]}} = \boldsymbol{a}^{[l-1]} \boldsymbol{\delta}^{[l]^T}.$$

Summary of backprop

- Calculate deltas at output units $\boldsymbol{\delta}^{[L]}$
- Backpropagate deltas through network using

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

- Calculate weight update

$$\Delta \boldsymbol{W}^{[l]} = -\eta \boldsymbol{\delta}^{[l]} (\boldsymbol{a}^{[l-1]})^T$$

The vanishing gradient problem

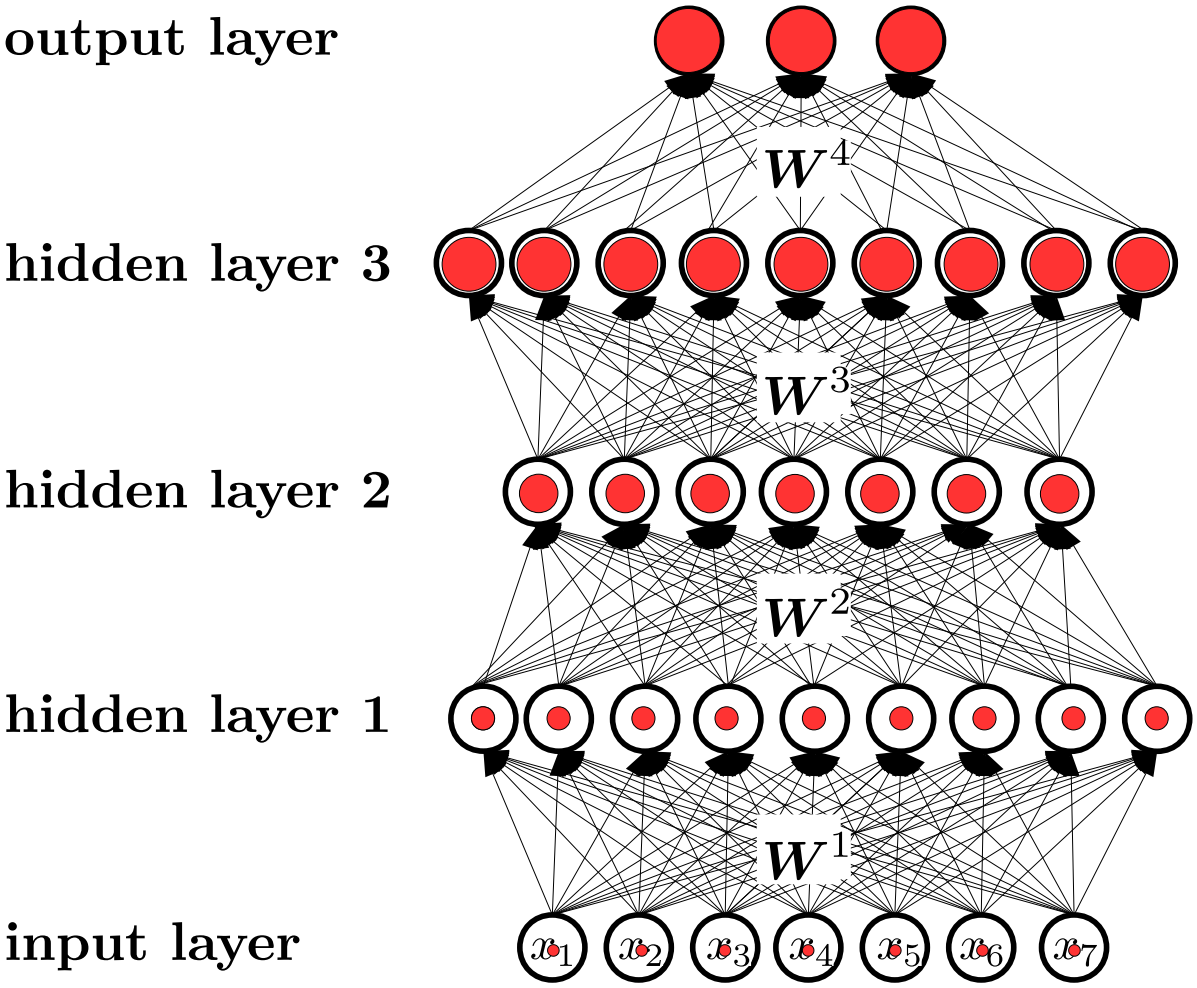
- We know from above:

$$\boldsymbol{\delta}^{[l-1]^T} = \boldsymbol{\delta}^{[l]^T} \underbrace{\boldsymbol{W}^{[l]} \operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]})\right)}_{\boldsymbol{J}^{[l]}} = \boldsymbol{\delta}^{[l]^T} \boldsymbol{J}^{[l]}$$

- Thus we have: $\|\boldsymbol{J}^{[l]}\| \leq k \leq 1$

$$\|\boldsymbol{\delta}^{[1]}\| = \|\boldsymbol{\delta}^{[L]^T} \prod_{l=L}^{2} \boldsymbol{J}^{[l]}\| \leq k^{L-1} \|\boldsymbol{\delta}^{[L]}\| \quad \Rightarrow \|\boldsymbol{\delta}^{[1]}\| \approx 0$$

The vanishing gradient problem



$$\boldsymbol{\delta}^{[L]}$$

$$\boldsymbol{\delta}^{[L-1]} = \boldsymbol{\delta}^{[L]} J^{[L]}$$

$$\boldsymbol{\delta}^{[L-2]} = \boldsymbol{\delta}^{[L-1]} J^{[L-1]}$$

$$\boldsymbol{\delta}^{[L-3]} = \boldsymbol{\delta}^{[L-2]} J^{[L-2]}$$

$$\vdots$$

$$\boldsymbol{\delta}^{[0]} = \boldsymbol{\delta}^{[1]} J^{[1]}$$

The vanishing gradient problem

Note: Vanishing and exploding gradients (Hochreiter, 1991)

From the recursion formula above (4.81)

$$\boldsymbol{\delta}^{[l-1]^T} = \boldsymbol{\delta}^{[l]^T} \boldsymbol{W}^{[l]} \operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]})\right),$$

we find an important property of neural network that influences the ability of a network to learn: the size (norm) of the delta-errors that backpropagated through the network. Since for each layer, the delta errors are multiplied by the Jacobian they exhibit an exponential behaviour (growth or shrinkage) across layers. The following principal options exist:

- $\|\boldsymbol{\delta}^{[l-1]}\| < \|\boldsymbol{\delta}^{[l]}\|$: **Vanishing gradients.** This has been the typical case since the derivatives of the sigmoid activation function is at most $|f'(x)| = 0.25$, the norm $\|\operatorname{diag}\left(f'(\boldsymbol{s}^{[l-1]})\right)\| \leq 0.25$ and the norm of the 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.** This would be the ideal case that the delta errors have a similar norm in each layer, and in fact many recent algorithmic improvements aim at keeping this quantity close to one. For example, initialization strategies, see Chapter 9.
- $\|\boldsymbol{\delta}^{[l-1]}\| > \|\boldsymbol{\delta}^{[l]}\|$: **Exploding gradients.** If the norm of the weight matrix is large, this scenario can occur, in which the norm of the delta-errors grow through each layer. Large weight updates lead to unstable learning or even numeric overflows.

Ameliorations of the VGP for feed-forward NN: Outlook

- Rectified linear units
- Initialization strategies
- Skip connections (ResNets)
- … (see later)

4.8 Jacobian

- The “Jacobian” of a neural network is an ambiguous term:

  1) The derivative of the loss function with respect to the parameters: $\frac{\partial L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{w}))}{\partial \boldsymbol{w}}$ “parameter Jacobian”

  2) The derivative of the network outputs with respect to the inputs: $\frac{\partial g(\boldsymbol{x}; \boldsymbol{w})}{\partial \boldsymbol{x}}$ “input-output Jacobian”

  3) The derivative of the outputs of a particular layer with respect to the layer inputs:

$$\frac{\partial}{\partial \boldsymbol{x}} f(\boldsymbol{W}\boldsymbol{x})$$

“single-layer Jacobian”

“**Jacobian**”

4.8 Jacobian: 1) parameter Jacobian

- The derivative of the loss function with respect to the parameters
  - Used for training neural networks with gradient descent

4.8 Jacobian: 2) input-output Jacobian

- The derivative of the network outputs with respect to the inputs.

$$\frac{\partial g(\boldsymbol{x}; \boldsymbol{w})}{\partial \boldsymbol{x}}$$

$$\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{x}}$$

Used to optimize input with respect to output (gradient ascent); “Deep Dream”

4.8 Jacobian: 3) (single-layer) Jacobian

- The derivative of the outputs of a particular layer with respect to the layer inputs:
  - Quantity that is connected to the VGP

$$\boldsymbol{a} = f(\boldsymbol{W}\boldsymbol{x})$$

$$\boldsymbol{J} = \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = \mathrm{diag}\left(f'(\boldsymbol{W}\boldsymbol{x})\right)\boldsymbol{W}$$

4.9 Hessian

- Matrix of second derivatives of the loss w.r.t. the weights:

$$\boldsymbol{H}_{ij} = \frac{\partial^2 L(\boldsymbol{y}, g(\boldsymbol{x}; \boldsymbol{w}))}{\partial w_i \; \partial w_j}$$

  where all parameters of the network are written as vector $\boldsymbol{w}$

- Used for (see later):
  - Optimization (?)
  - Pruning
  - Error bars for predictions
  - Second order Taylor approximation

$$R(\boldsymbol{w}) = R(\boldsymbol{w}_0) + (\boldsymbol{w} - \boldsymbol{w}_0)^T \boldsymbol{g} + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_0)^T \boldsymbol{H} (\boldsymbol{w} - \boldsymbol{w}_0) + \mathcal{O}\left((\boldsymbol{w} - \boldsymbol{w}_0)^3\right)$$

4.10 Efficient Training of DNNs and MLPs: Basic tricks of the trade

- Online, stochastic and batch training
- Sampling from training data, batch size, epochs and learning curves
- Normalizing inputs and outputs
- Initialization
- Learning rates
- Number of neurons and hidden layers

Training, validation and test set

- (sloppy) definitions:
  - *Training set* (e.g. 80% of data): set of input objects that are presented to the network; weights are adjusted to fit this set
  - *Validation set* (e.g. 10% of data): set of input objects on which the network is checked; withheld from the network; network architecture and hyperparams are adjusted on this set
  - *Test set* (e.g. 10% of data): set of input objects on which the final trained network is checked; withheld from network and from architecture/hyperparameter selection
- More on that: Chapter 5 “Generalization error, test set”

Online, stochastic and batch training

- Online learning/training: single sample from training set is propagated through network and then a parameter update is performed.

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \nabla_{\boldsymbol{w}} R_{\text{emp}}(y, \boldsymbol{x}, \boldsymbol{w})|_{\boldsymbol{w}^{\text{old}}}$$

- Full-batch learning/training: all samples from training set are propagated through network and then a parameter update is performed.

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w})|_{\boldsymbol{w}^{\text{old}}}$$

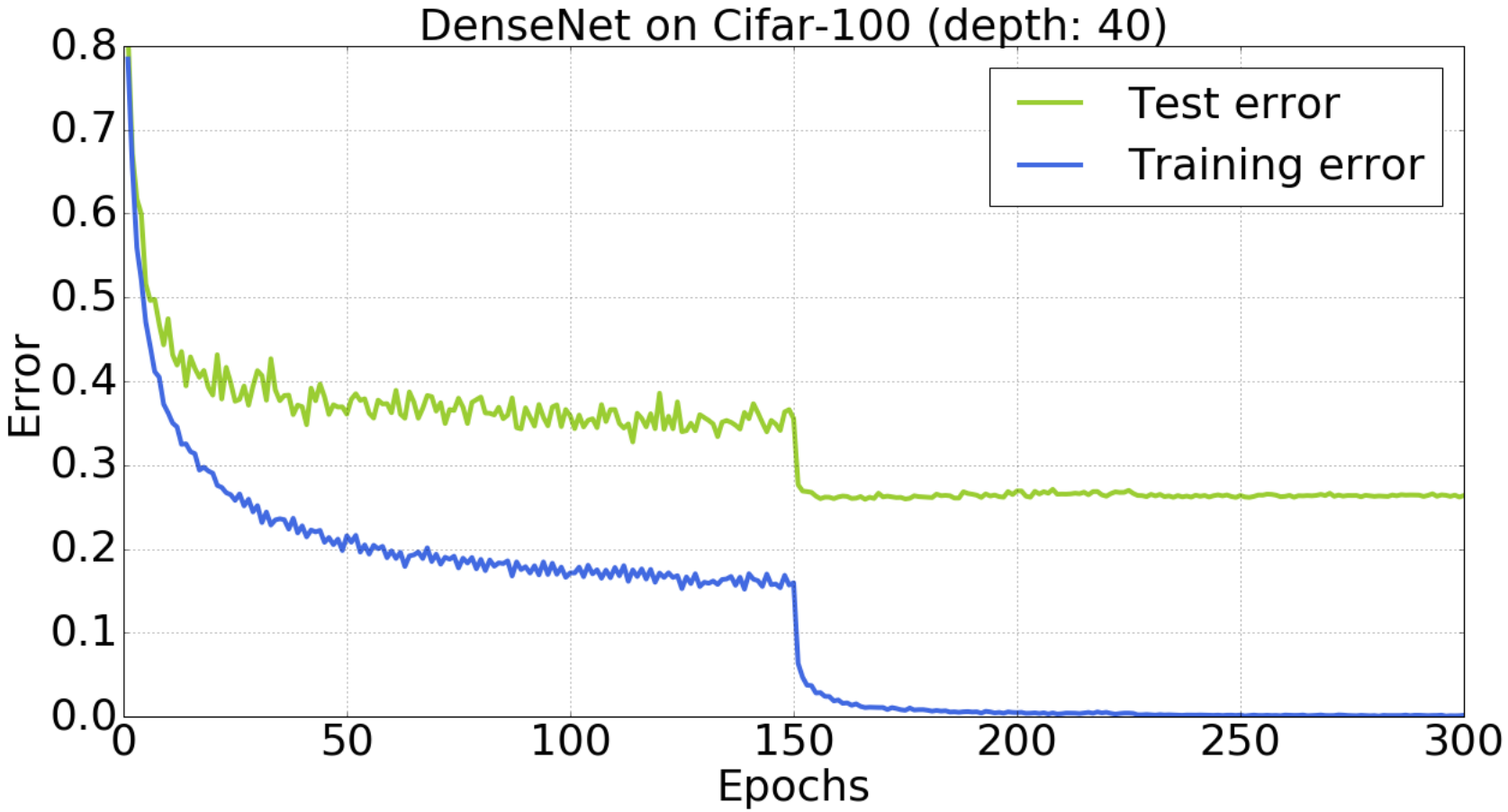
- Stochastic learning/training: a small subset of samples from the training set are propagetd through network and then a parameter update is performed

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} - \eta \tilde{\nabla}_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w})|_{\boldsymbol{w}^{\text{old}}}$$

Approx. the full-batch gradient with a random subsample (“mini batch”):

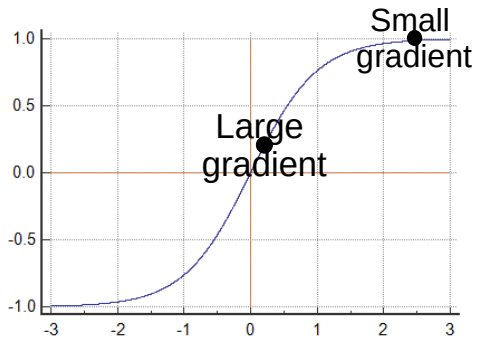
$$\nabla_{\boldsymbol{w}} R_{\text{emp}}(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{w}} L(y^n, g(\boldsymbol{x}^n, \boldsymbol{w})) \approx \frac{1}{B} \sum_{b=1}^{B} \nabla_{\boldsymbol{w}} L(y^{n_b}, g(\boldsymbol{x}^{n_b}, \boldsymbol{w}))$$

Epochs, batch size, learning curves



Normalizing inputs and outputs

- Inputs should be normalized to zero mean and unit variance
  - Calculate mean and variance of each feature on training set to perform normalization
  - Training is faster in this case:



- Categorical features: one-hot encoding

Number of neurons and hidden layers

- Depends on complexity of the problem to solve
  - More complex → more layers and neurons
- Rule of thumb:
  Use as many layers and neurons as needed such that the network can overfit to the training data. Then add regularization to prevent overfitting
- What is *overfitting*?
  - See next lecture.

Initializiation

- Weights should be initialized with zero mean
  - Gaussian, Uniform or truncated Gaussian
- Variance:
  - $\frac{2}{J}$ for ReLU activations
  - $\frac{1}{J}$ else
- Later: chapter “Initialization”

Learning rates

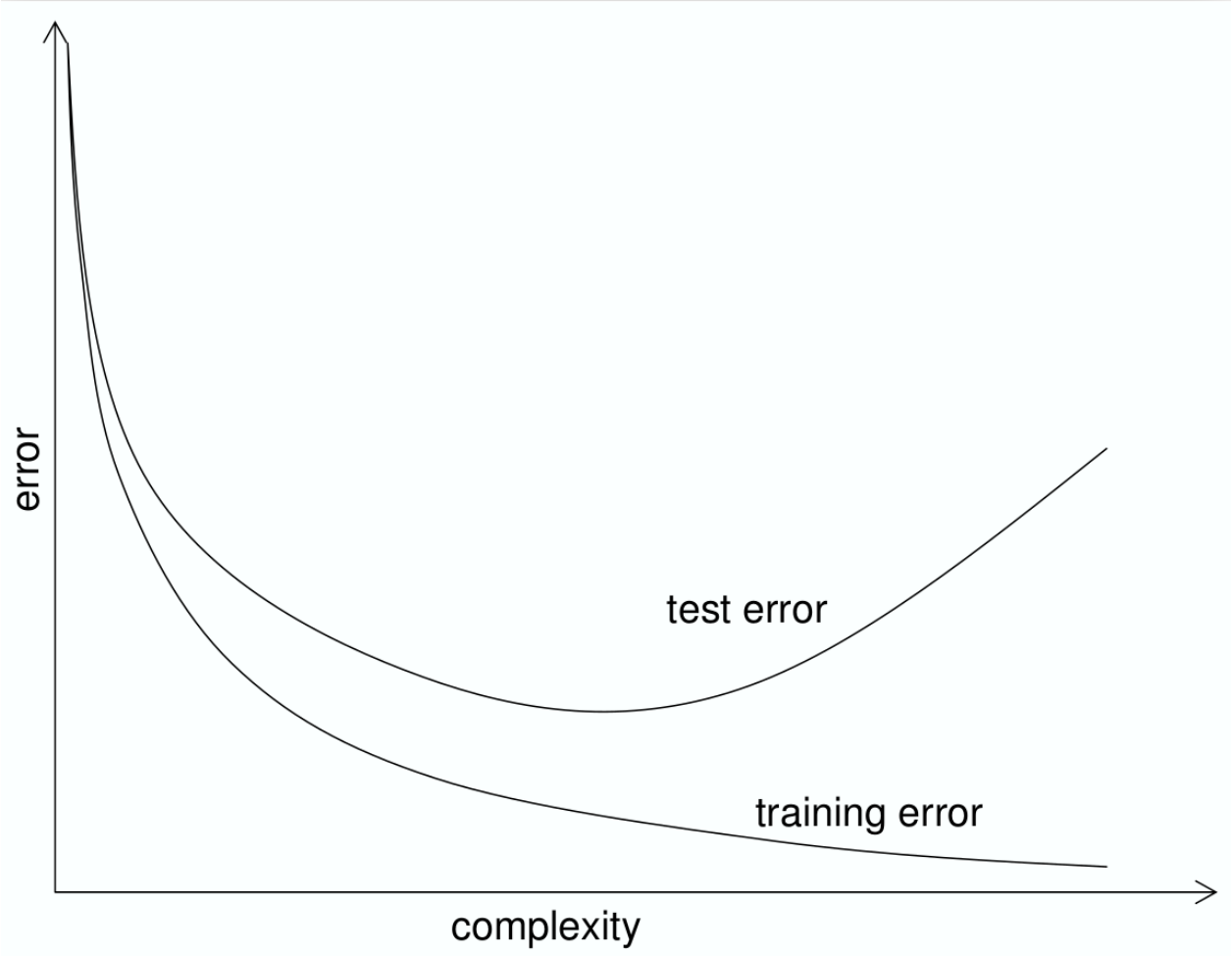
- One of the most important hyperparameters
  - Typical values 0.1, 0.01
  - Search in this space: $10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, \ldots$

- Heuristics: Start training with learning rate as high as possible without divergence of the network; decrease later;

Number of neurons and hidden layers

- Depends on complexity of the problem to solve
  - More complex → more layers and neurons
- Rule of thumb:
  Use as many layers and neurons as needed such that the network can overfit to the training data. Then add regularization to prevent overfitting
- What is *overfitting*?
  - See next lecture.

Outlook: generalization



Summary

- Discussed forward and backward pass of and MLP and of a deep FNN
- Main problem with deep nets: VGP
- Jacobian, Hessian
- Tricks of the trade for training FNNs