Deep Learning for NLP

Lecture 3 – Learning in MLPs / Backprop

Dr. Steffen Eger
Wei Zhao
Niraj Pandey



Natural Language Learning Group (NLLG)
Technische Universität Darmstadt

This lecture:

- Gradient Descent Learning:
  - A general optimization technique
- Backpropagation
  - A general technique to determine the gradients in *all* neural networks
- Language Modeling

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Gradient Descent

NLLG

Excursion: Continuous Optimization

- Consider generally the problem

$$\min_{\mathbf{w}\in\mathbb{R}^n} F(\mathbf{w})$$

for a smooth function $F : \mathbb{R}^n \rightarrow \mathbb{R}.$

Excursion: Continuous Optimization

- Consider generally the problem

$$\min_{\mathbf{w}\in\mathbb{R}^n} F(\mathbf{w})$$

  for a smooth function $F : \mathbb{R}^n \to \mathbb{R}.$

- One **general technique** for addressing this problem is *gradient descent*:

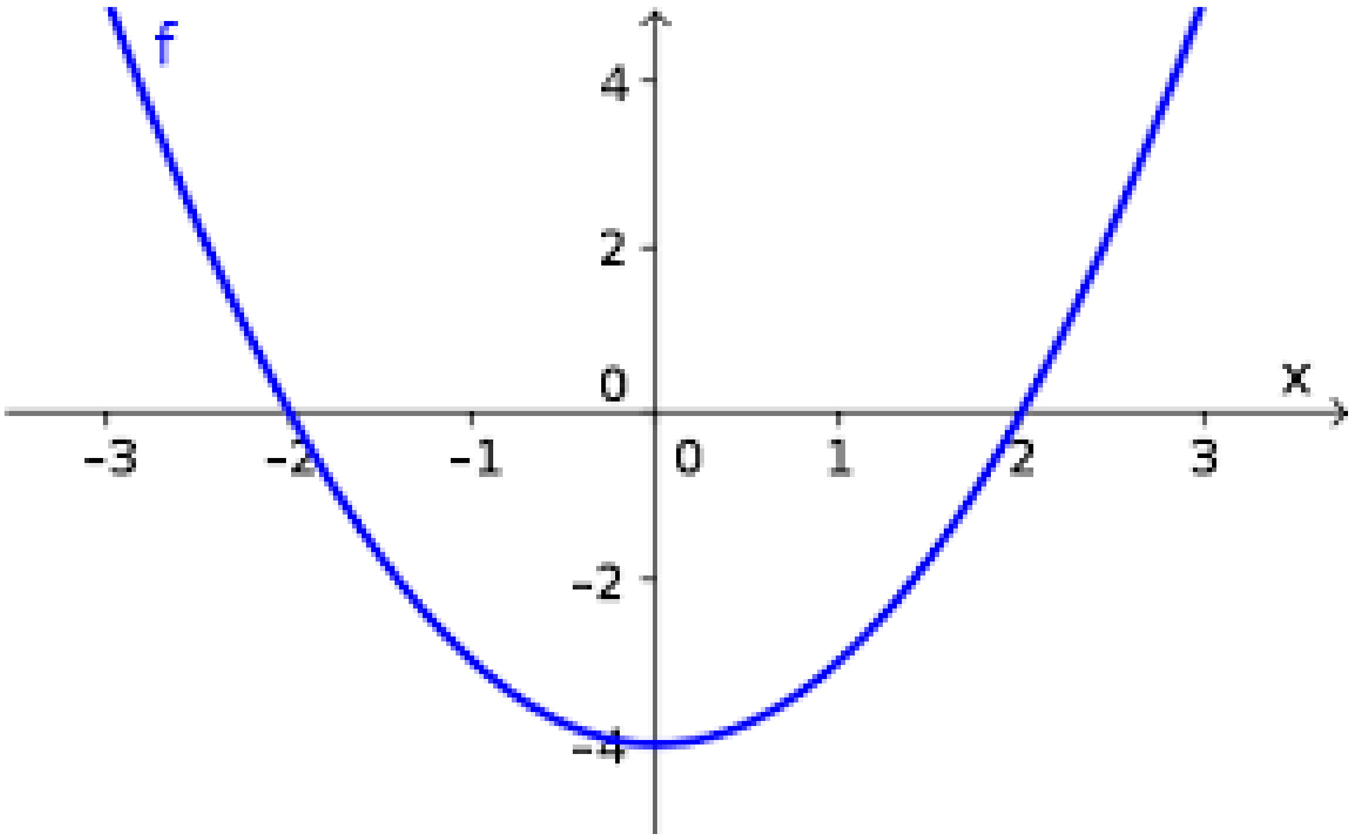
$$\mathbf{w}' \leftarrow \mathbf{w} - \alpha \nabla F(\mathbf{w})$$

  where $\alpha > 0$ and $\nabla F(\mathbf{w})$ is the *gradient* of F, evaluated at $\mathbf{w}$:

$$\nabla F(\mathbf{w}) = \begin{pmatrix} \frac{\partial F}{\partial y_1}(\mathbf{w}) \\ \vdots \\ \frac{\partial F}{\partial y_n}(\mathbf{w}) \end{pmatrix}$$

Gradient descent by example

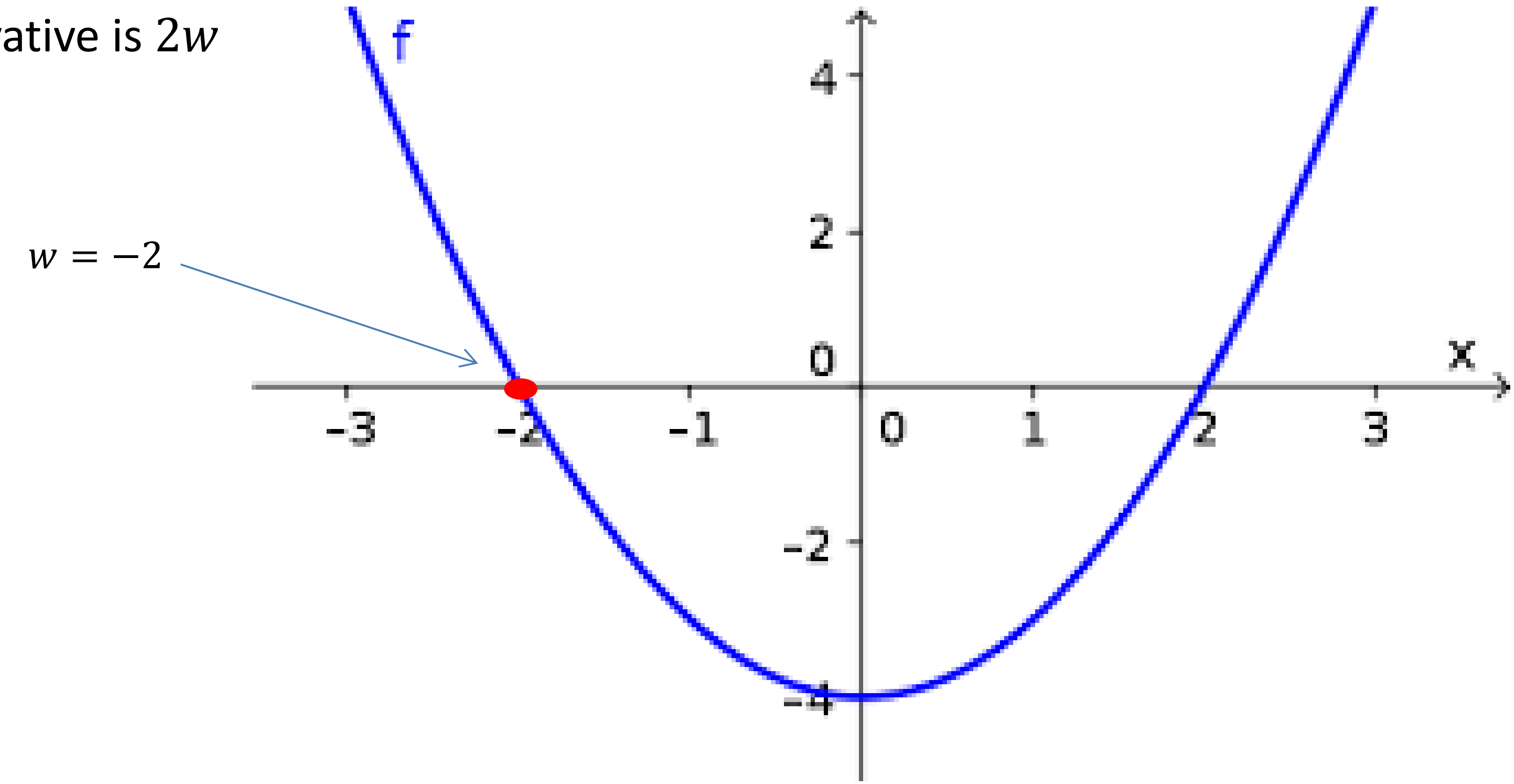
- Consider $\mathrm{F(w)} = w^2 - 4$
- Derivative is $2w$



From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent by example

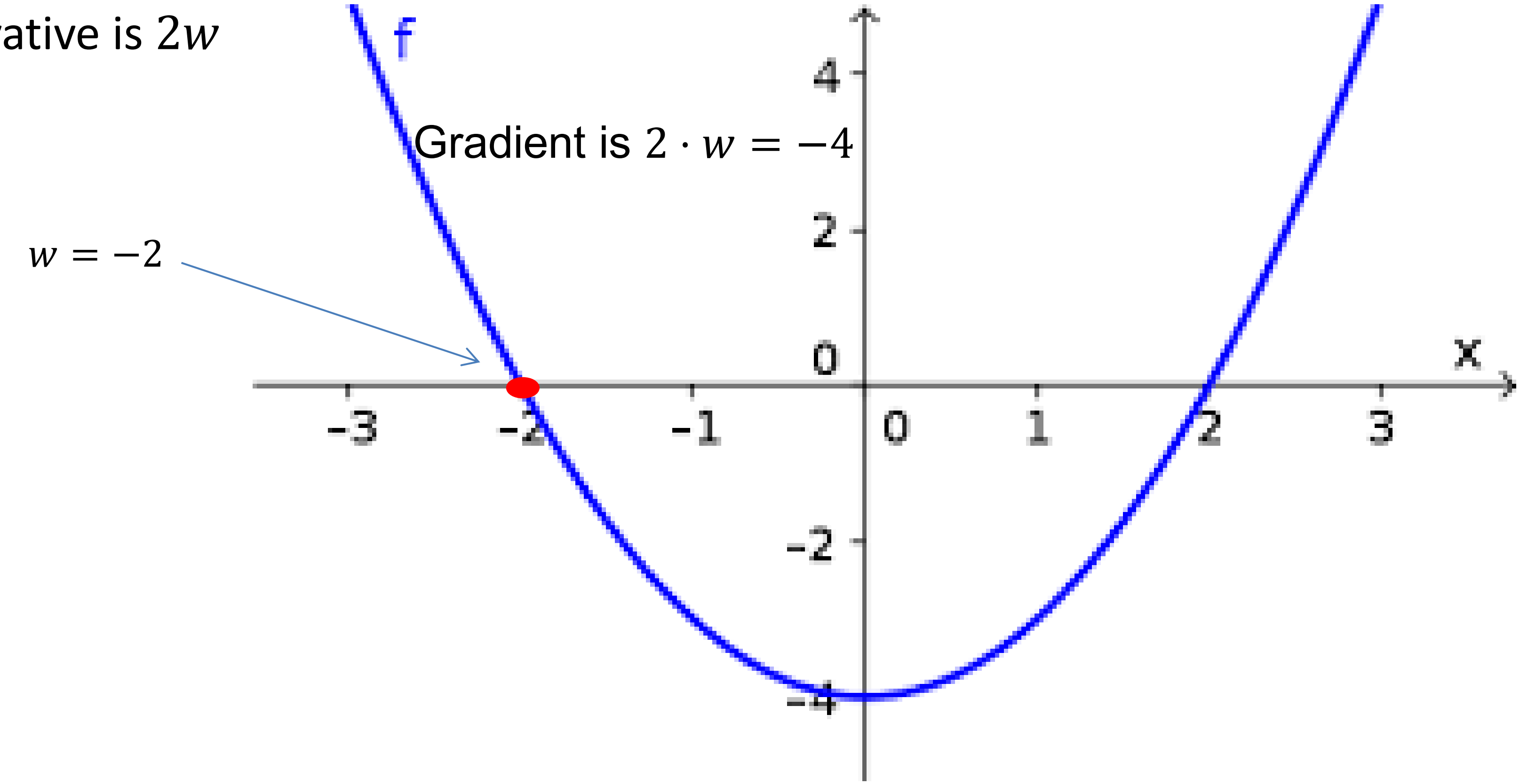
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent by example

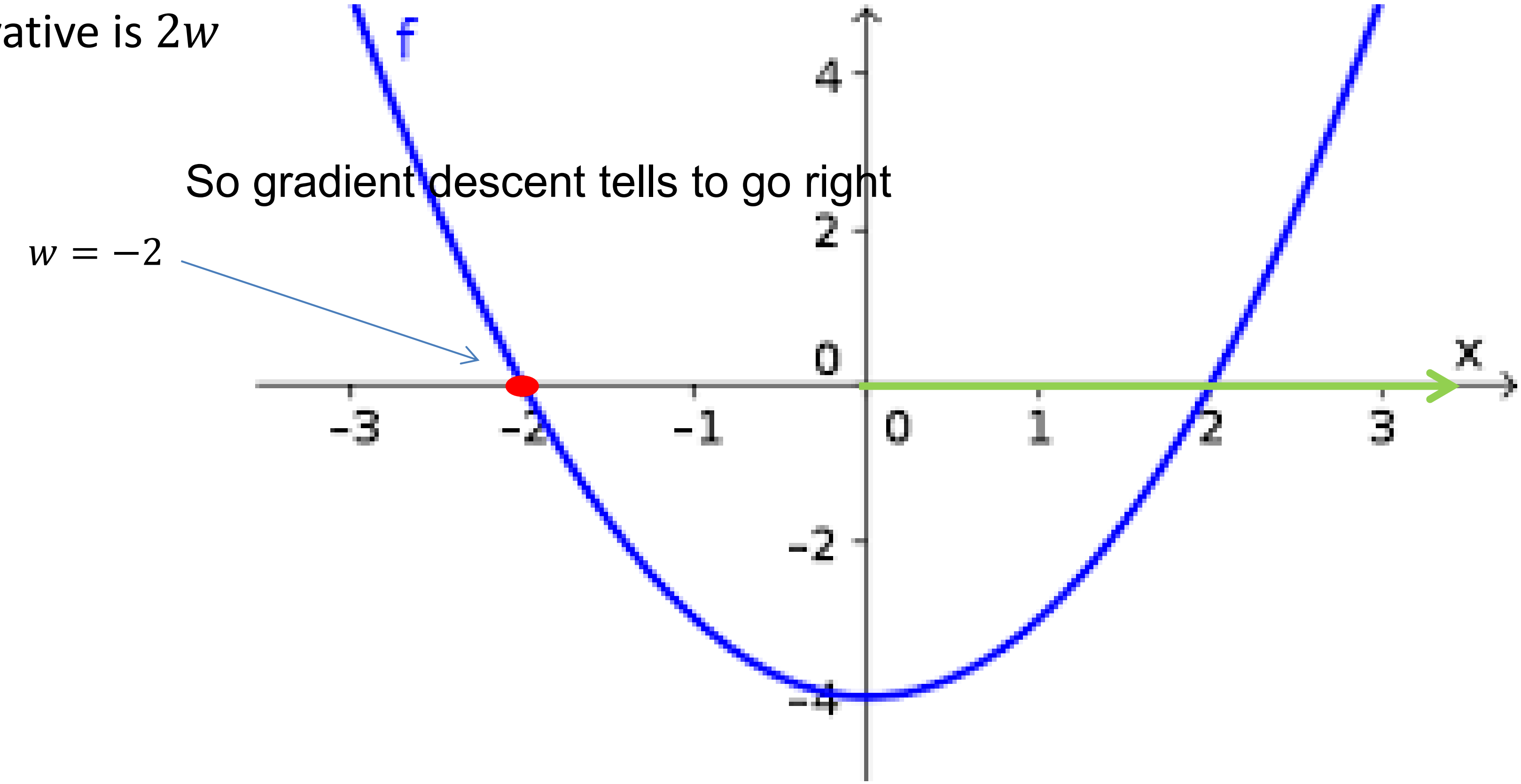
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent by example

- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



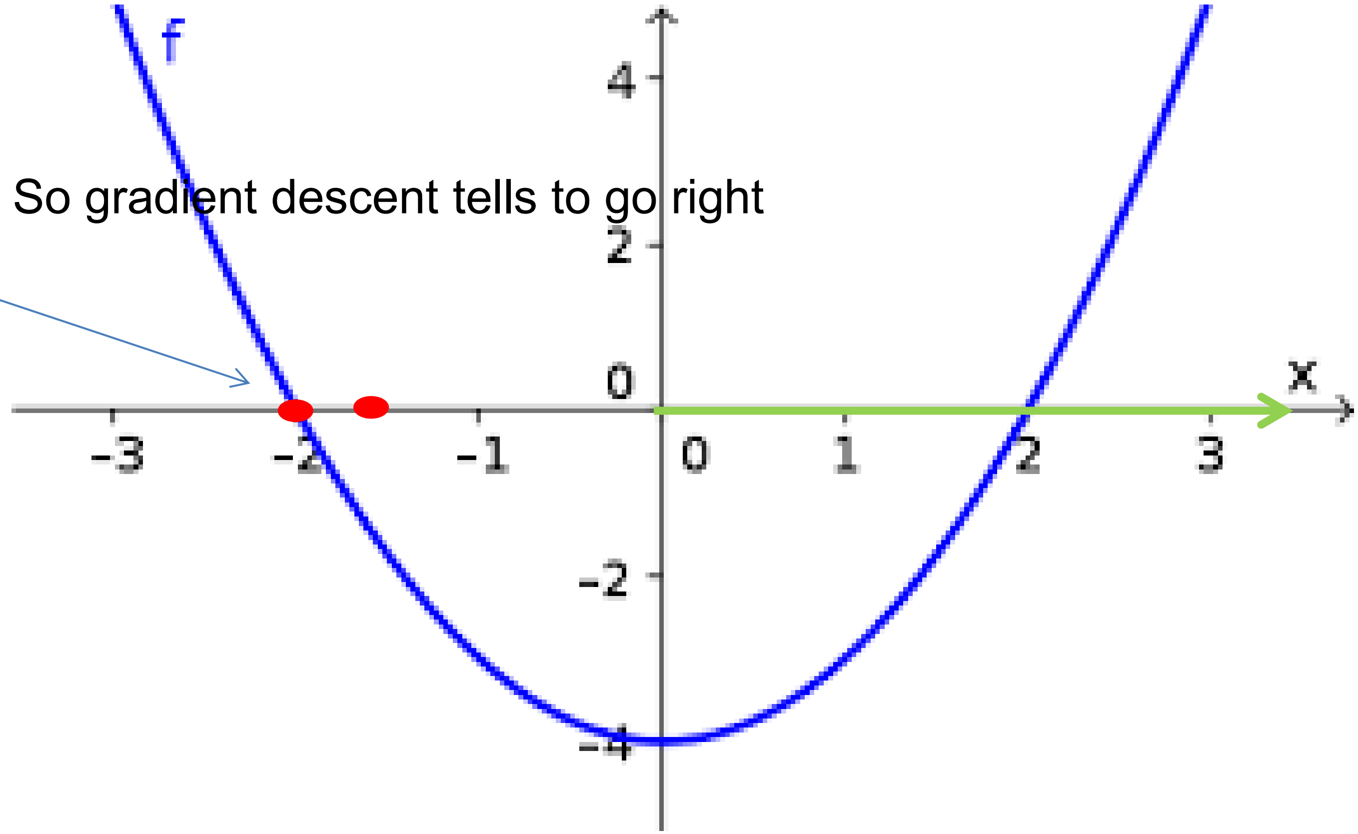
From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent by example

- Consider $\mathrm{F(w)} = w^2 - 4$
- Derivative is $2w$

So gradient descent tells to go right

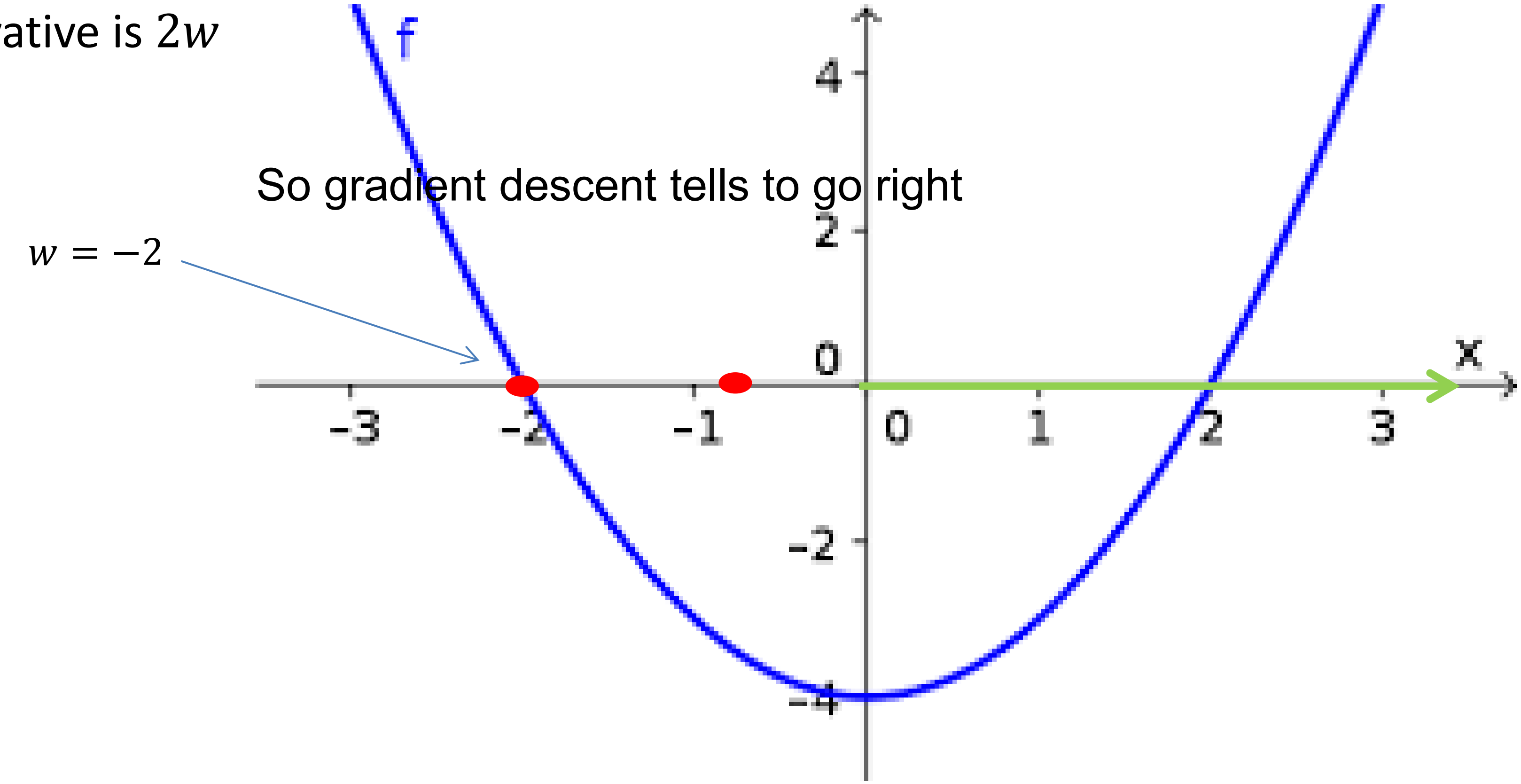
$w = -2$



From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent by example

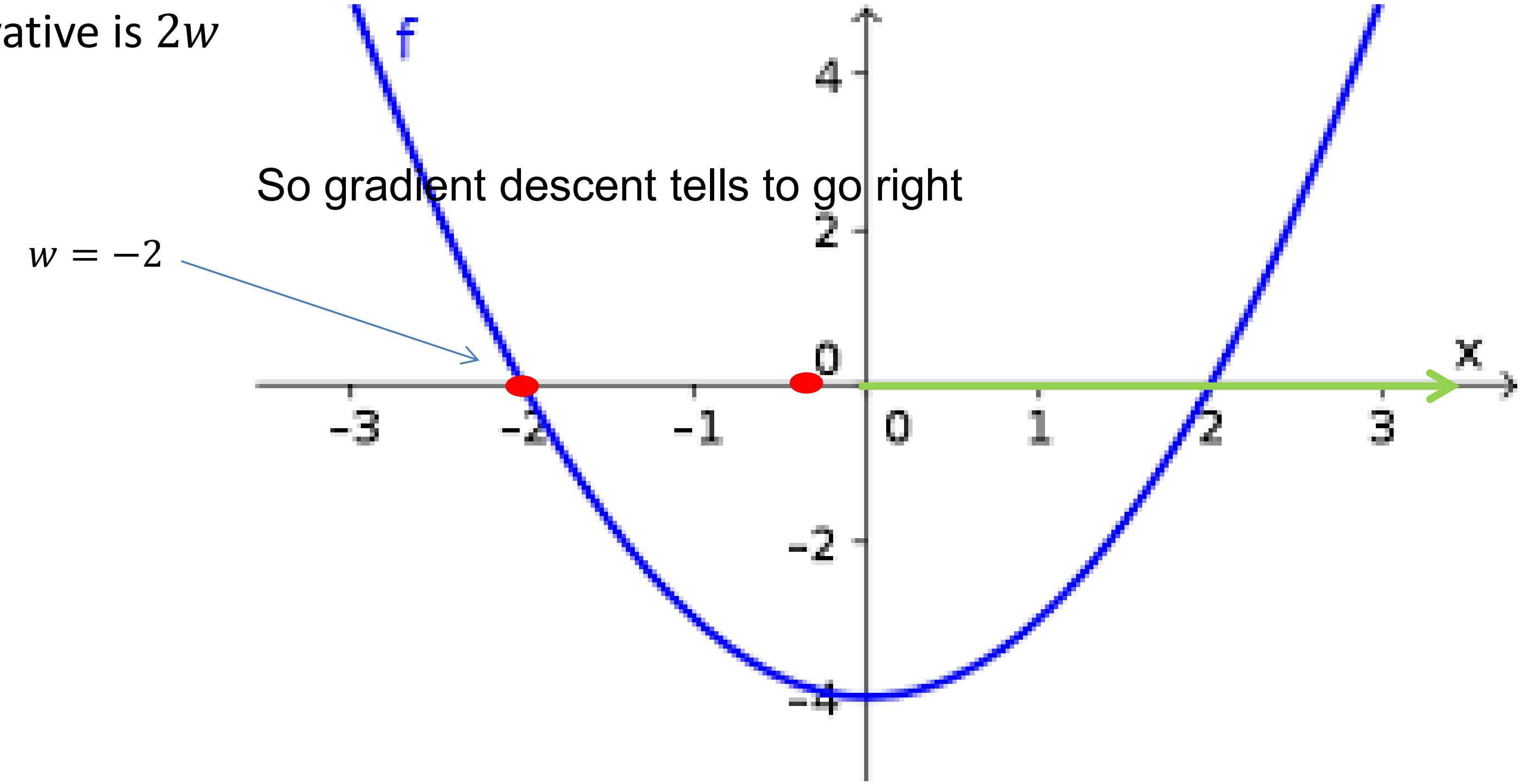
- Consider $\mathrm{F(w)} = w^2 - 4$
- Derivative is $2w$



From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent by example

- Consider $\mathrm{F(w)} = w^2 - 4$
- Derivative is $2w$



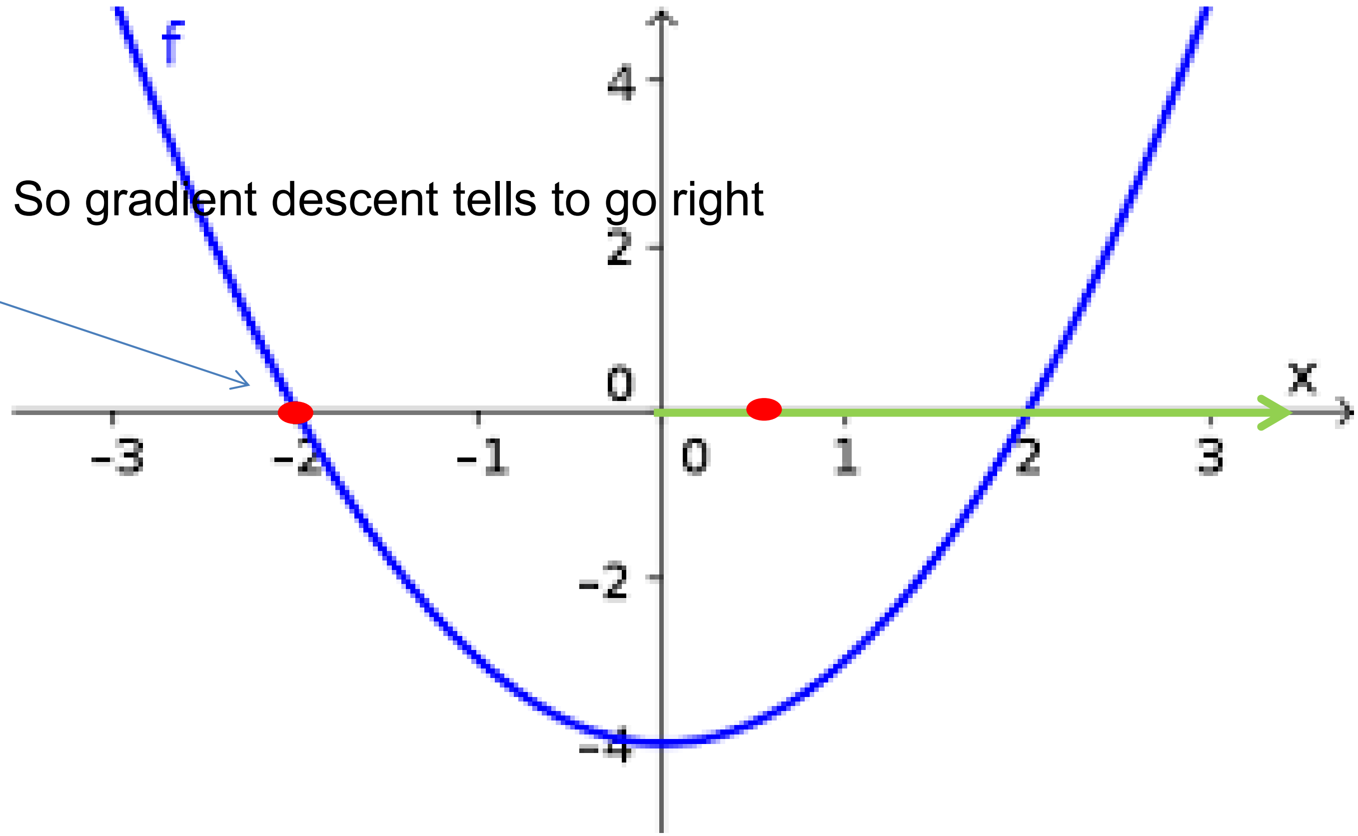
From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent by example

- Consider $\mathrm{F(w)} = w^2 - 4$
- Derivative is $2w$

So gradient descent tells to go right

$w = -2$

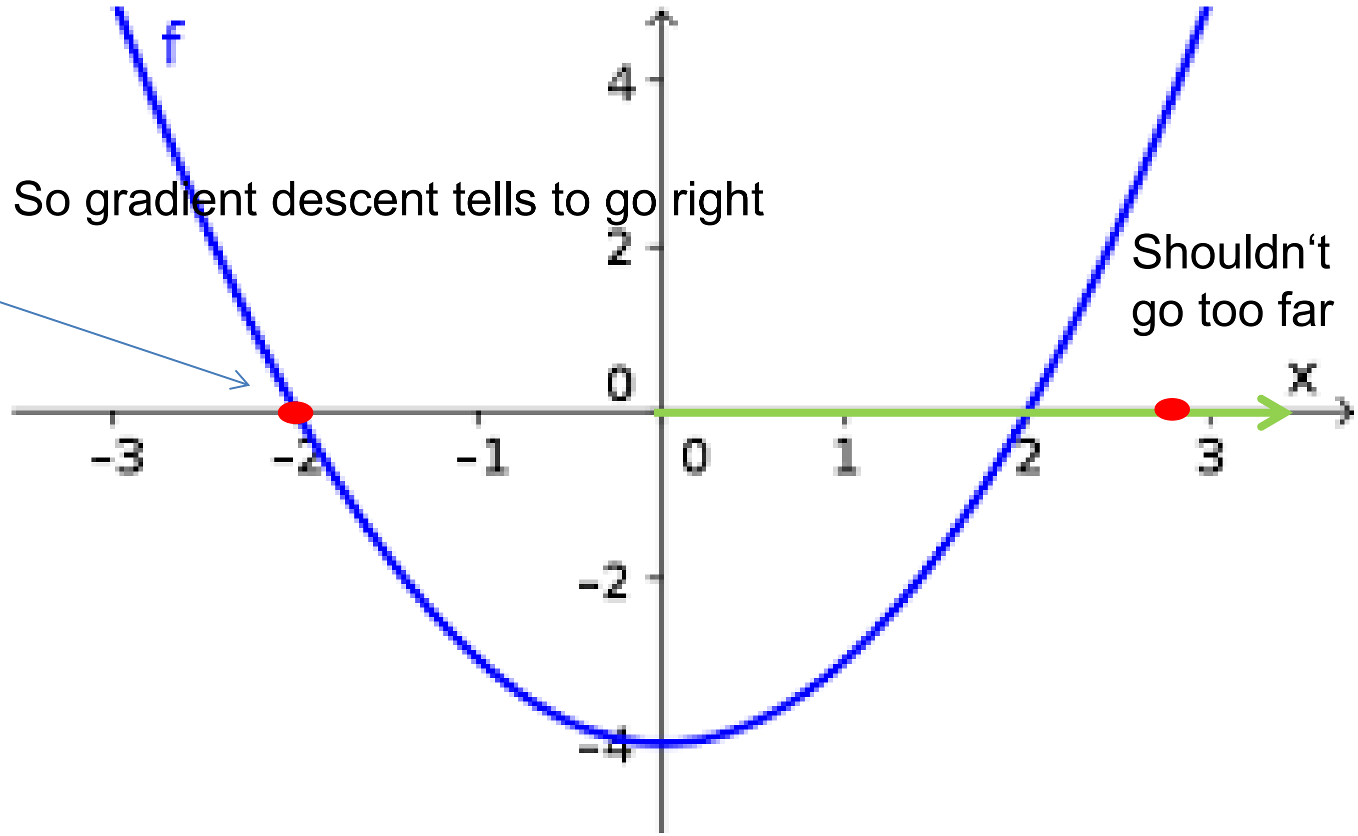


From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent by example

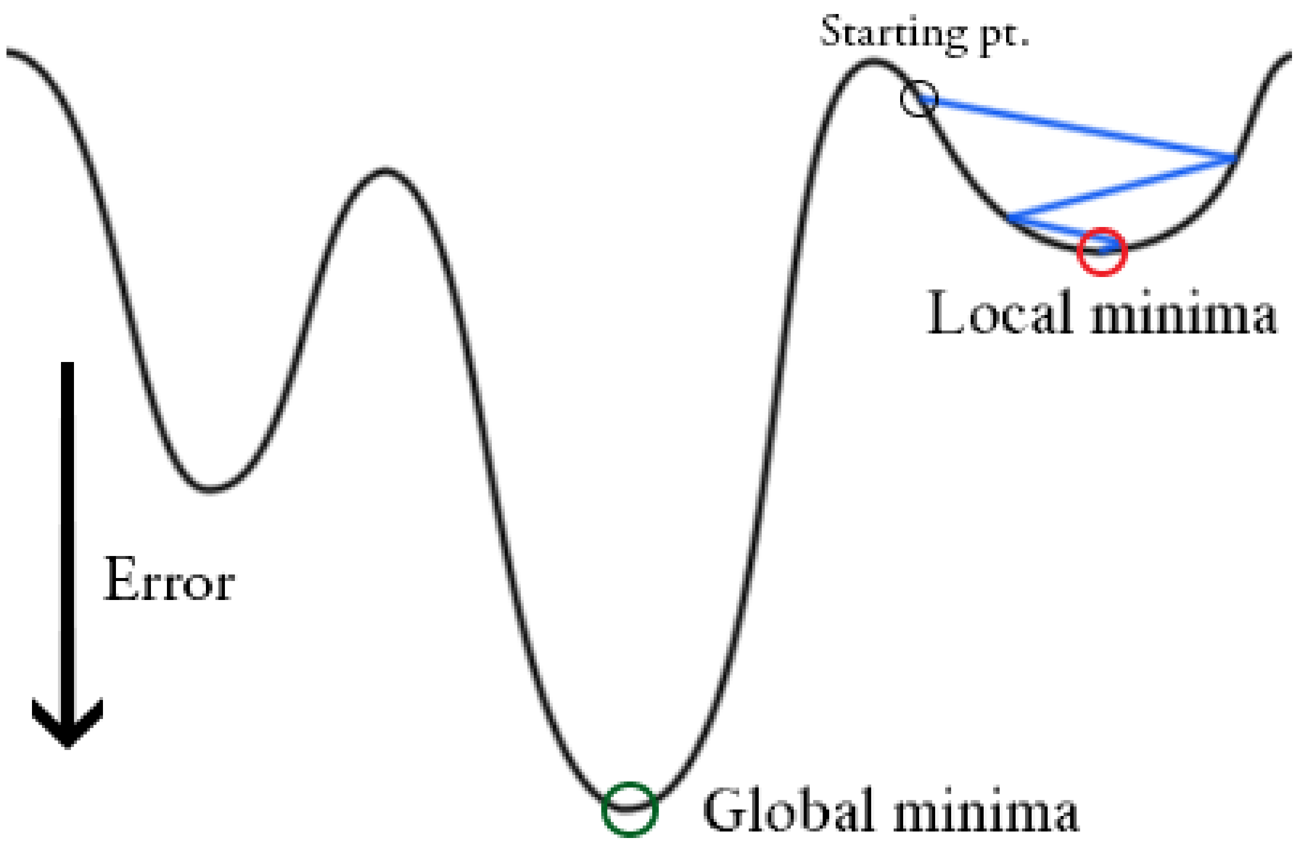
- Consider $\mathrm{F(w)} = w^2 - 4$
- Derivative is $2w$

$w = -2$



From http://mathinsight.org/media/image/image/graph_x_squared_minus_4.png; note that this is $x^2 - 4$

Gradient descent does not always lead to good solutions



From https://static.thinkingandcomputing.com/2014/03/bprop.png

Other optimization techniques

- Note that other optimization techniques exist
  - Newton methods (2nd order, Hessian)
  - Conjugate gradient
  - ….
- They may converge faster or be guaranteed to find global optima
  - May also require stronger assumptions
  - Second order methods need to determine the matrix of 2nd order derivatives
  - Often difficult to compute / rarely used for training neural networks

Batch Learning

- **Given data:** $(\boldsymbol{x}_1, \boldsymbol{t}_1), \ldots, (\boldsymbol{x}_n, \boldsymbol{t}_n)$
- **Error- / Loss-Function:** $\ell(\boldsymbol{y}, \boldsymbol{t})$
- **Objective:**

$$\min_{\boldsymbol{w}} F(\boldsymbol{w}) = \min_{\boldsymbol{w}} \sum_i \ell(f(\boldsymbol{x}_i; \boldsymbol{w}), \boldsymbol{t}_i)$$

where f is (e.g.) a neural network

$\boldsymbol{w} = \boldsymbol{\theta}$ stands for any set of parameters

- **Batch learning:**
  - **Compute gradient based on all datapoints**

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla \mathrm{F}(\mathbf{w})$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \sum_i \nabla \ell(f(\boldsymbol{x}_i; \boldsymbol{w}), \boldsymbol{t}_i)$$

Online Learning

- **Given data:** $(\boldsymbol{x}_1, \boldsymbol{t}_1), \ldots, (\boldsymbol{x}_n, \boldsymbol{t}_n)$
- **Error- / Loss-Function:** $\ell(\boldsymbol{y}, \boldsymbol{t})$
- **Objective:**

$$\min_{\boldsymbol{w}} F(\boldsymbol{w}) = \min_{\boldsymbol{w}} \sum_i \ell(f(\boldsymbol{x}_i; \boldsymbol{w}), \boldsymbol{t}_i)$$

where f is (e.g.) a neural network

- **Online learning:**
  - **Approximate** $\nabla \mathrm{F}$ by computing it at only one data point

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla \ell(f(\boldsymbol{x}_i; \boldsymbol{w}), \boldsymbol{t}_i) \qquad \text{for all } i = 1, \ldots, n$$

Mini-Batch Learning

- Can be seen as intermediate solution between batch and online learning
- Select $k < n$ datapoints (randomly), compute the loss function, update weights
- $k = 1 \rightarrow$ online learning

- *Advantage:*
  - Computing loss function gradient on all datapoints can be computationally expensive
  - Mini-batch learning converges faster to a good solution than batch learning

- Unclear how to choose k
  - Smaller k's lead often to better solutions (generalize better)
  - Larger k's are computationally advantageous on networks trained on multiple machines

- Mini-batch learning is also known as (a.k.a) **stochastic gradient descent**

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Beyond gradient descent

NLLG

Modifications

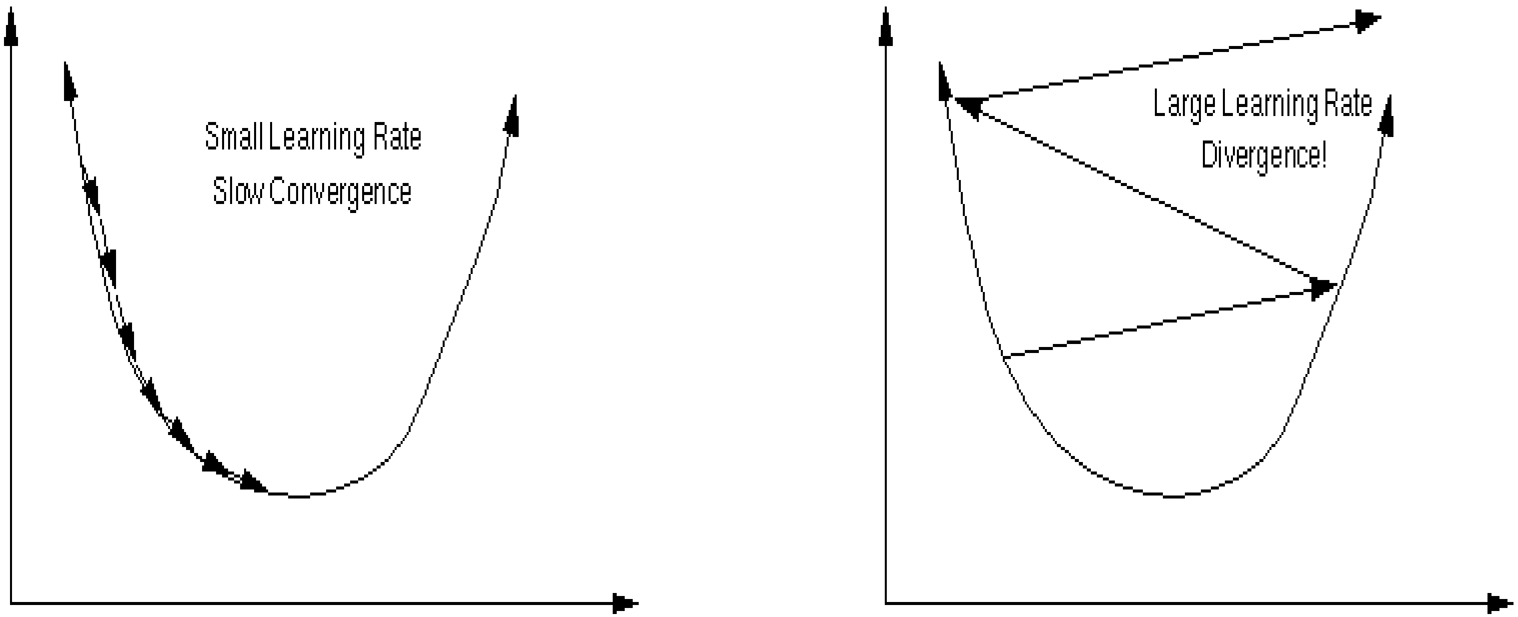
- Adapt learning rate α over time: $\alpha \rightarrow \alpha_t$

- Include a *regularization term:*
  - $\min_{\{\boldsymbol{w} \in R^n\}} F(\boldsymbol{w}) + \gamma \left|\left|\boldsymbol{w}\right|\right|^2$

- Include *momentum*
  - smooth with past gradients

Modifications: Learning rate

- Adapt learning rate α over time: $\alpha \rightarrow \alpha_t$



From https://www.willamette.edu/~gorr/classes/cs449/precond.html

Modifications: Regularization

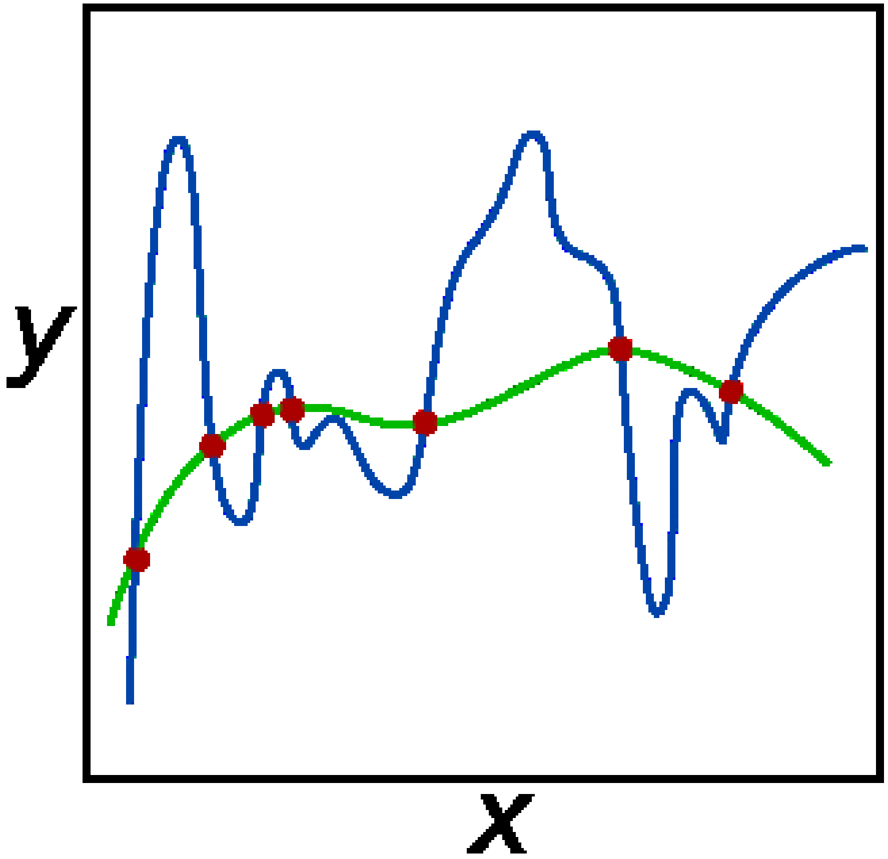
TECHNISCHE UNIVERSITÄT DARMSTADT

- Minimize
  - $\min_{\{\boldsymbol{w} \in R^n\}} F(\boldsymbol{w}) + \gamma R(\boldsymbol{w})$
  - for $\gamma \geq 0$

- Choose, e.g.,
  - $R(\boldsymbol{w}) = ||\boldsymbol{w}||^2 = \sum_i w_i^2$ (L2 regularization)
  - $R(\boldsymbol{w}) = |\boldsymbol{w}| = \sum_i |w_i|$ (L1 regularization)

- Motivation: Occam‘s razor (choose simpler solutions over more complicated ones)

Modifications: Regularization



- From https://en.wikipedia.org/wiki/Regularization_(mathematics)

Advanced Optimizers

- Stochastic gradient descent (SGD) has troubles:
  - Ravines: Surface is more steep on one direction
  - Saddle points and plateaus
- More advanced optimizers have been proposed:
  - RMSProp, AdaGrad, AdaDelta, Adam, Nadam
- These methods train usually faster than SGD
- Found solution is often not as good as by SGD
  - Possible solution: First train with Adam, fine-tune with SGD
- Great overview: http://ruder.io/optimizing-gradient-descent/index.htm

Recommendations

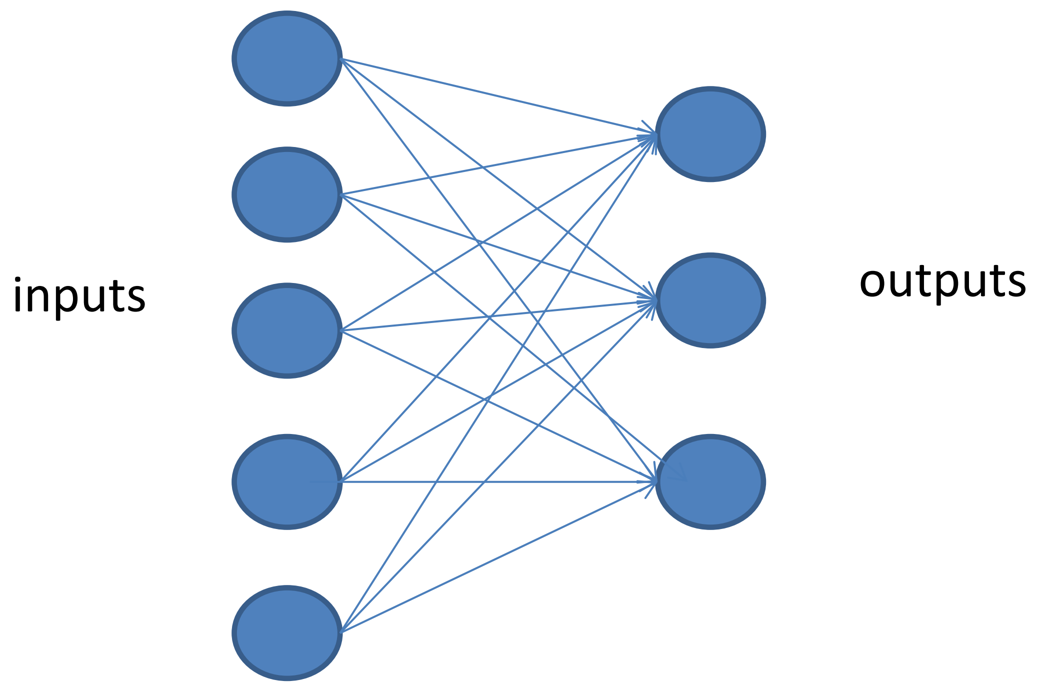
- Try out Adam, Adagrad, Nesterov momentum, standard momentum
  - Often standardly built-in in libraries such as Tensorflow, Keras, etc.
- Use regularization

Outline

Deep Networks – Terminology (Recap)

More complex neural networks

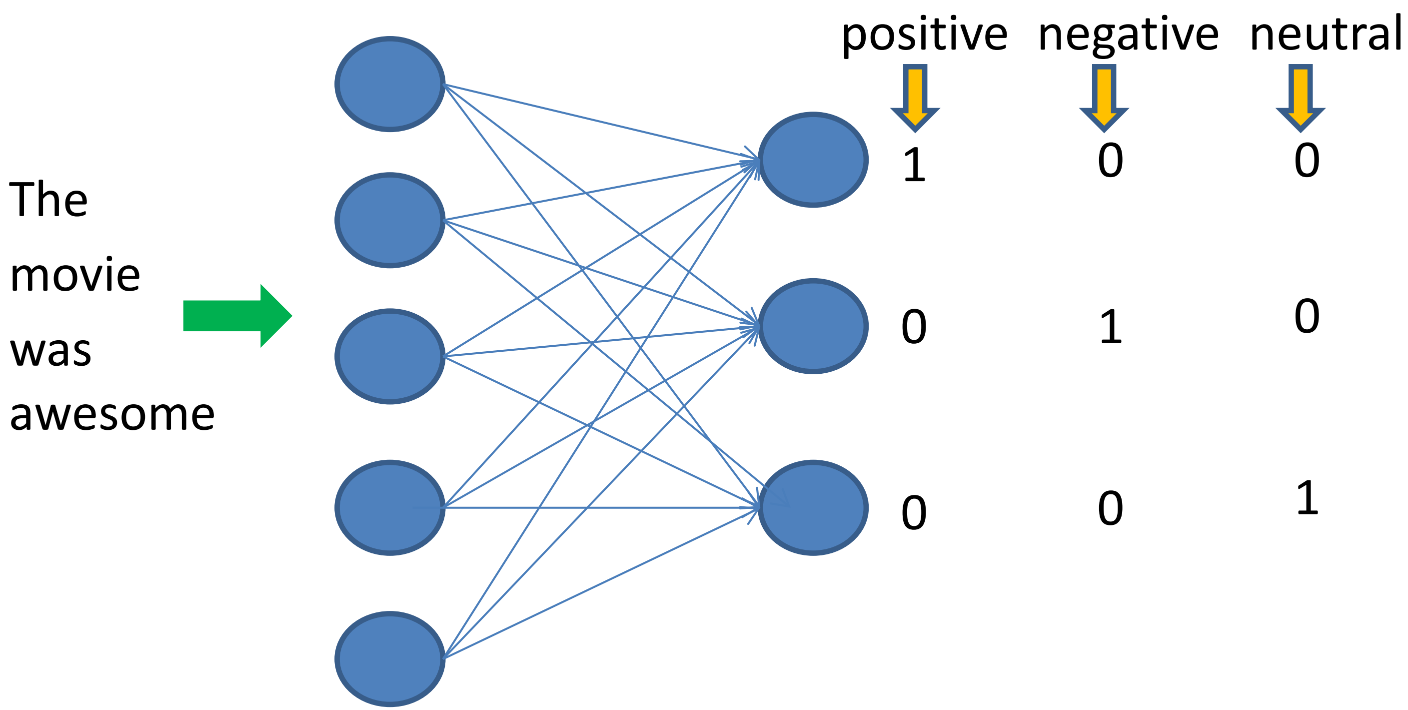
- Several output neurons instead of a single output neuron



- No additional technical difficulty, can use the previous learning techniques

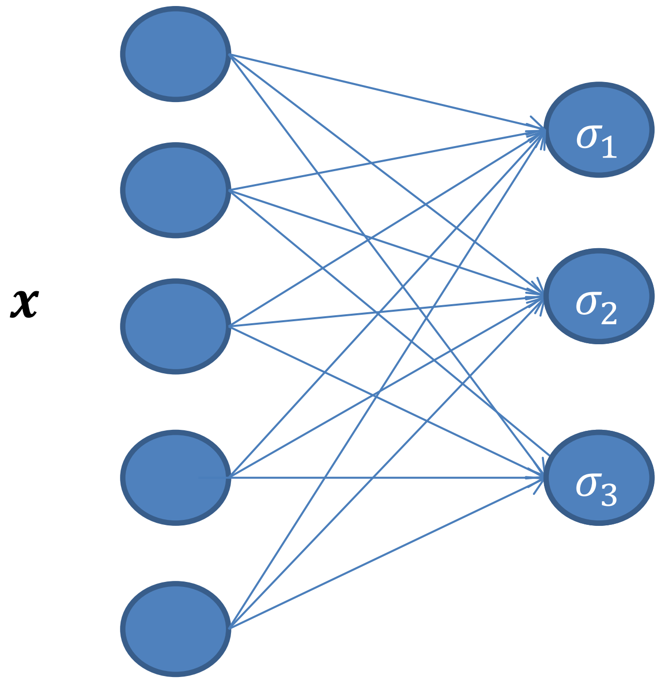
More complex neural networks

- Several output neurons instead of a single output neuron



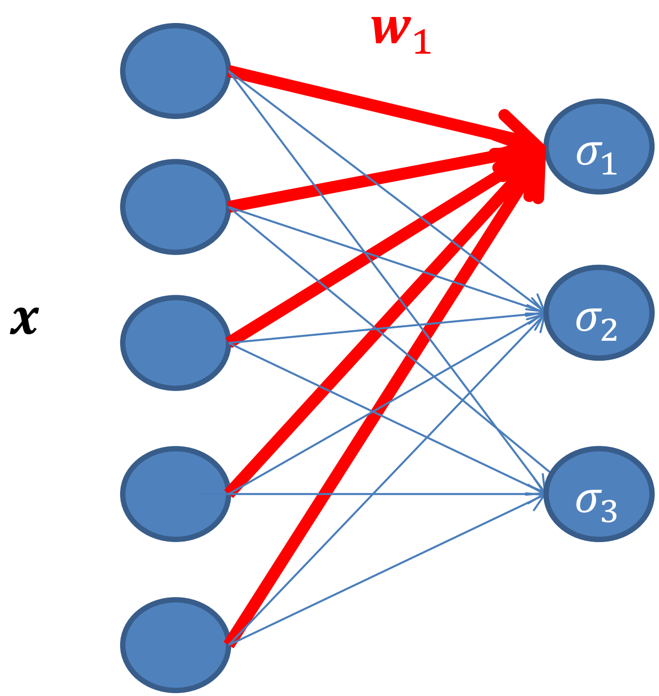
More complex neural networks

- Several output neurons instead of a single output neuron



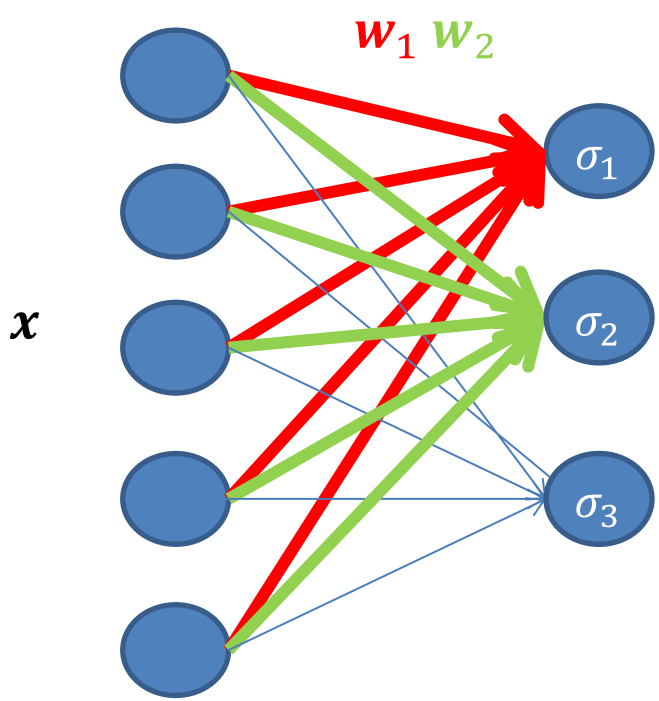
More complex neural networks

- Several output neurons instead of a single output neuron



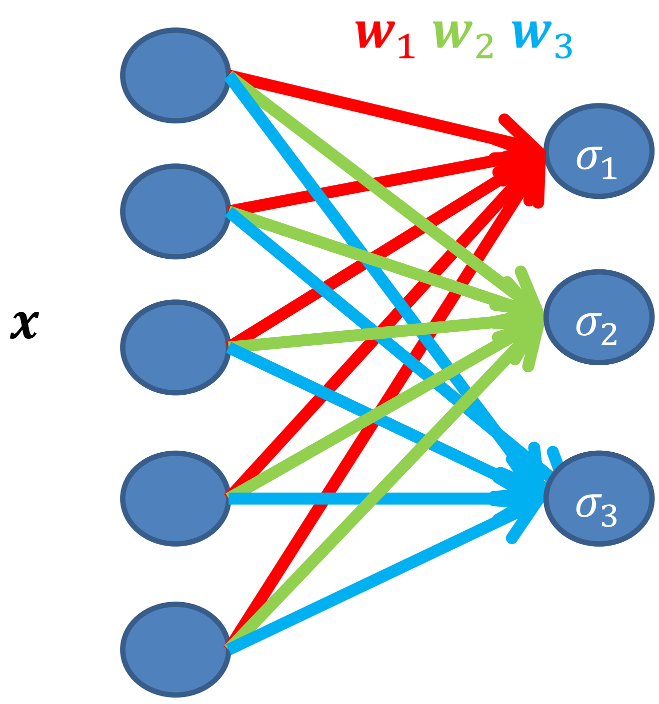
More complex neural networks

- Several output neurons instead of a single output neuron



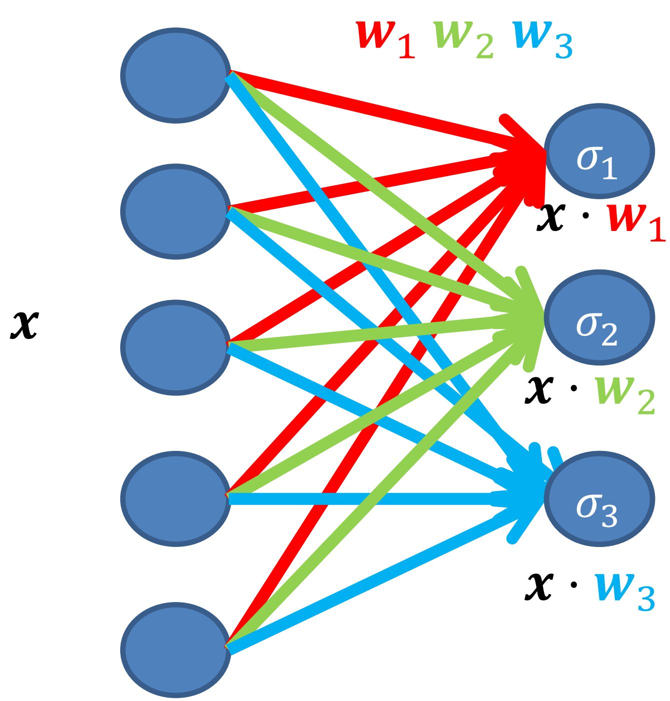
More complex neural networks

- Several output neurons instead of a single output neuron



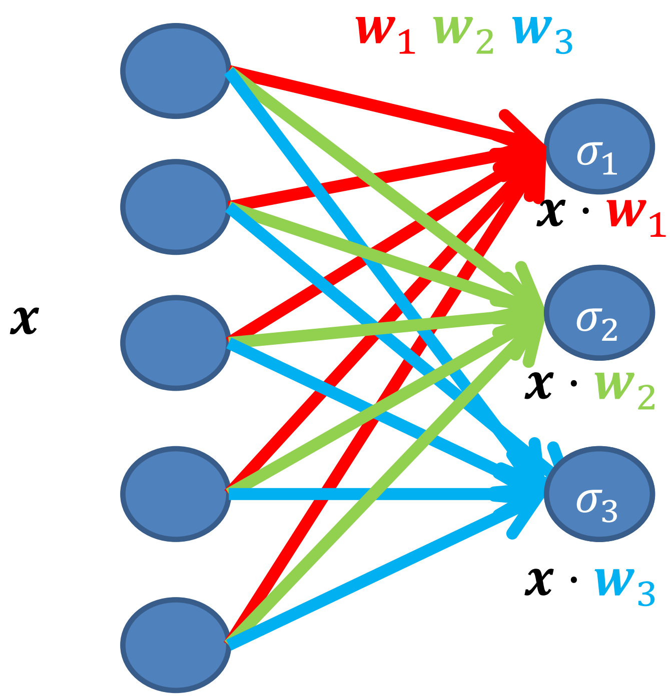
More complex neural networks

- Several output neurons instead of a single output neuron



More complex neural networks

- Several output neurons instead of a single output neuron



$\sigma_1(x \cdot w_1)$

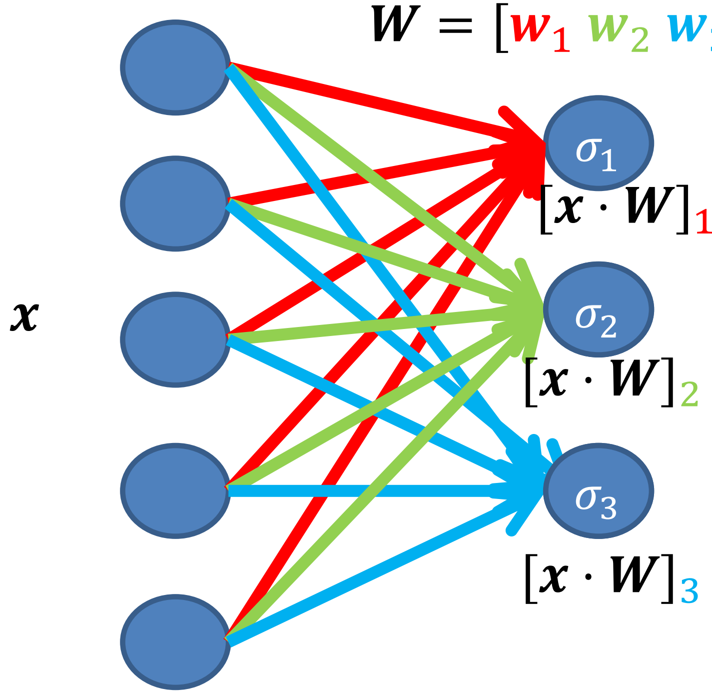
$\sigma_2(x \cdot w_2)$

$\sigma_3(x \cdot w_3)$

More complex neural networks

- Several output neurons instead of a single output neuron

$$W = [w_1\ w_2\ w_3], W \in R^{n\times m}, x \in R^{1\times n}$$



$$[\sigma_1(x \cdot W)]_1$$

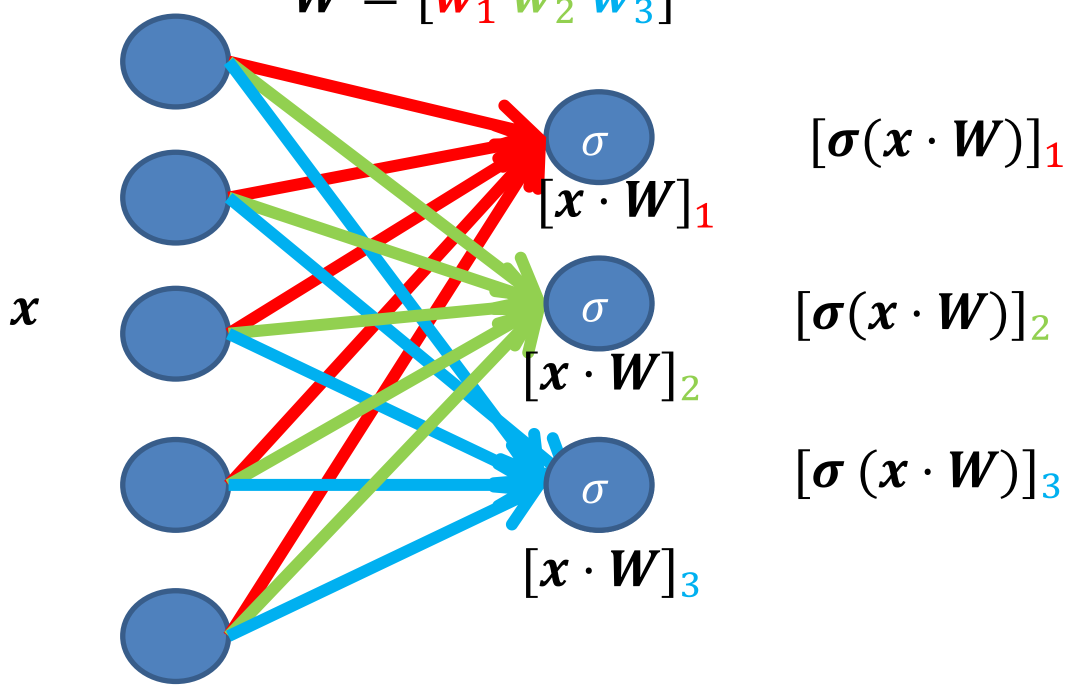
$$[\sigma_2(x \cdot W)]_2$$

$$[\sigma_3(x \cdot W)]_3$$

More complex neural networks

- Several output neurons instead of a single output neuron

$$W = [w_1\ w_2\ w_3]$$



A formal description: Multiple Output units

- **Network with n input units, m output units, and no hidden layers**
- **Given**
  - **Weight vectors** $\boldsymbol{W} = [\boldsymbol{w}_1 \cdots \boldsymbol{w}_m]$ **,** each $\boldsymbol{w}_k \in R^n$, so $\boldsymbol{W} \in R^{n \times m}$
  - **Non-linearities** $\boldsymbol{\sigma}_1, \ldots, \boldsymbol{\sigma}_m : \boldsymbol{R} \to \boldsymbol{R}$
- **Input to network**
  - **Input vector** $\mathbf{x} \in \mathbb{R}^{1 \times n}$, for $n \geq 1$. (recall that one input is reserved for bias)
- **Output units**
  - **Inputs (pre-activation):** $\boldsymbol{x} \cdot \boldsymbol{w}_1, \ \ldots, \ \boldsymbol{x} \cdot \boldsymbol{w}_m$
  - **Outputs:** $\boldsymbol{\sigma}_1(\boldsymbol{x} \cdot \boldsymbol{w}_1), \ldots, \boldsymbol{\sigma}_m(\boldsymbol{x} \cdot \boldsymbol{w}_m)$

A formal description: Multiple Output units

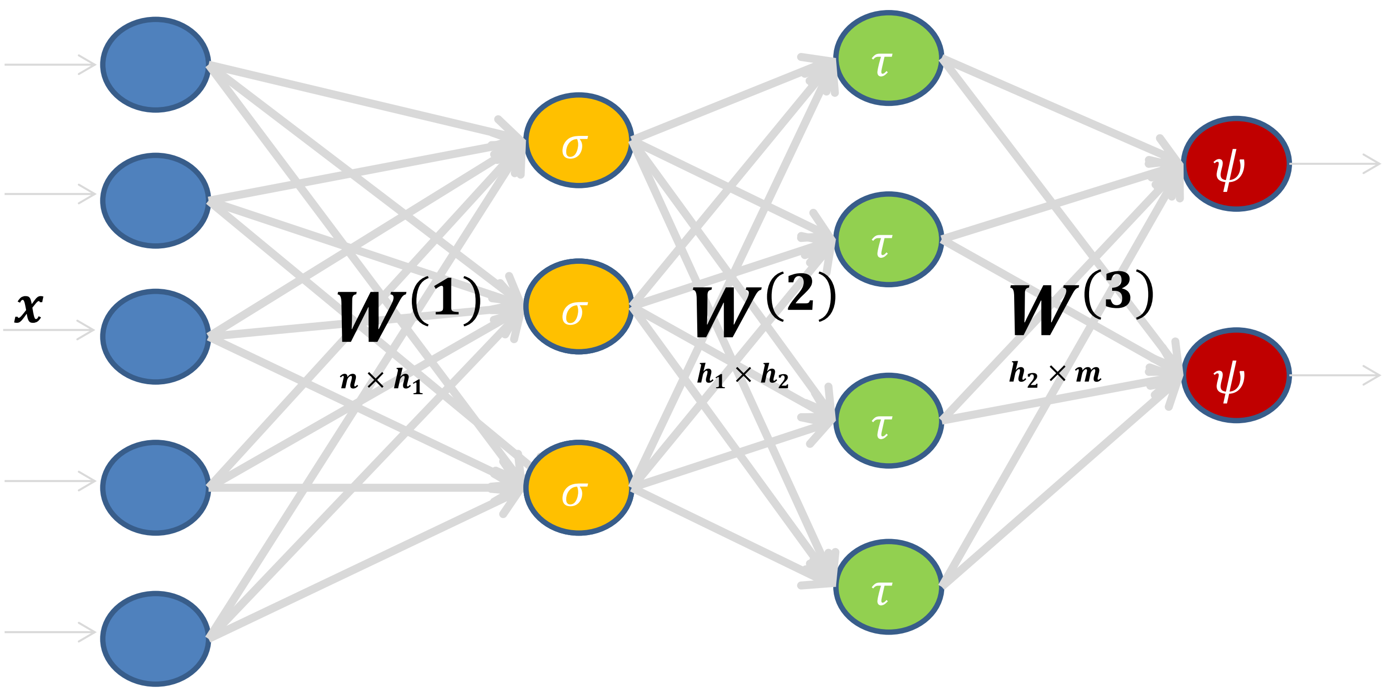
TECHNISCHE UNIVERSITÄT DARMSTADT

- **Network with n input units, m output units, and no hidden layers**
- **Given**
  - **Weight vectors** $\boldsymbol{W} = [\boldsymbol{w}_1 \cdots \boldsymbol{w}_m]$, each $\boldsymbol{w}_k \in R^{n\times 1}$, so $\boldsymbol{W} \in R^{n\times m}$
  - **Non-linearity** $\boldsymbol{\sigma}: \boldsymbol{R} \rightarrow \boldsymbol{R}$
- **Input to network**
  - **Input vector** $\mathbf{x} \in \mathbb{R}^{1\times n}$, for $n \geq 1$. (recall that one input is reserved for bias)
- **Output units**
  - **Inputs (pre-activation):** $\boldsymbol{x} \cdot \boldsymbol{W} \in \boldsymbol{R}^{\boldsymbol{1}\times \boldsymbol{m}}$
  - **Outputs:** $\boldsymbol{\sigma}(\boldsymbol{x} \cdot \boldsymbol{W}) = [\boldsymbol{\sigma}(\boldsymbol{x} \cdot \boldsymbol{w}_1), \ldots, \boldsymbol{\sigma}(\boldsymbol{x} \cdot \boldsymbol{w}_m)]$

    (Here, we apply the non-linearity element-wise)

More complex neural networks

- Multiple **hidden layers/**units



- This is called Multi-Layer-Perceptron (MLP). More difficult in terms of optimization

More complex neural networks

- Formally, a feed-forward neural network with H hidden units is a function
  - $f: \boldsymbol{R}^n \rightarrow \boldsymbol{R}^m$,
  - with parameter (matrices) $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \dots, \boldsymbol{W}^{(H)}$
  - and non-linearities $\sigma_1, \sigma_2, \dots, \sigma_H$
  - where
  - $\boldsymbol{z}^{(1)} = \boldsymbol{x} \cdot \boldsymbol{W}^{(1)}$
  - $\boldsymbol{y}^{(1)} = \sigma_1(\boldsymbol{z}^{(1)})$
  - $\boldsymbol{z}^{(2)} = \boldsymbol{y}^{(1)} \cdot \boldsymbol{W}^{(2)}$
  - .....
  - $\boldsymbol{z}^{(H)} = \boldsymbol{y}^{(H-1)} \cdot \boldsymbol{W}^{(H)}$
  - $\boldsymbol{y} = \boldsymbol{y}^{(H)} = \sigma_H(\boldsymbol{z}^{(H)})$

Feed-forward NN
a.k.a.
MLP

More complex neural networks

- Formally, a feed-forward neural network with H hidden units is a function
  - $f: \boldsymbol{R}^n \to \boldsymbol{R}^m$,
  - with parameter (matrices) $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \dots, \boldsymbol{W}^{(H)}$ and biases $\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \dots, \boldsymbol{b}^{(H)}$
  - and non-linearities $\sigma_1, \; \sigma_2, \quad \dots, \sigma_H$
  - where
  - $\boldsymbol{z}^{(1)} = \boldsymbol{x} \cdot \boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}$
  - $\boldsymbol{y}^{(1)} = \sigma_1(\boldsymbol{z}^{(1)})$
  - $\boldsymbol{z}^{(2)} = \boldsymbol{y}^{(1)} \cdot \boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}$
  - .....
  - $\boldsymbol{z}^{(H)} = \boldsymbol{y}^{(H-1)} \cdot \boldsymbol{W}^{(H)} + \boldsymbol{b}^{(H)}$
  - $\boldsymbol{y} = \sigma_H(\boldsymbol{z}^{(H)})$

Why do we need hidden layers?

- Hidden layers can learn useful intermediate representations of the data
  - Helps learning
  - A good organization of hidden layers can make learning much faster

- Perceptron cannot even learn the XOR function
  - In contrast, MLP with one hidden layer is a *universal approximator*,
    - See Cybenko 1989, Approximations by superpositions of sigmoidal functions; Hornik (1991), Approximation Capabilities of Multilayer Feedforward Networks
    - i.e. can represent/approximate any continuous function
    - More hidden layers can still be useful for learning an actual task

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Backpropagation

NLLG

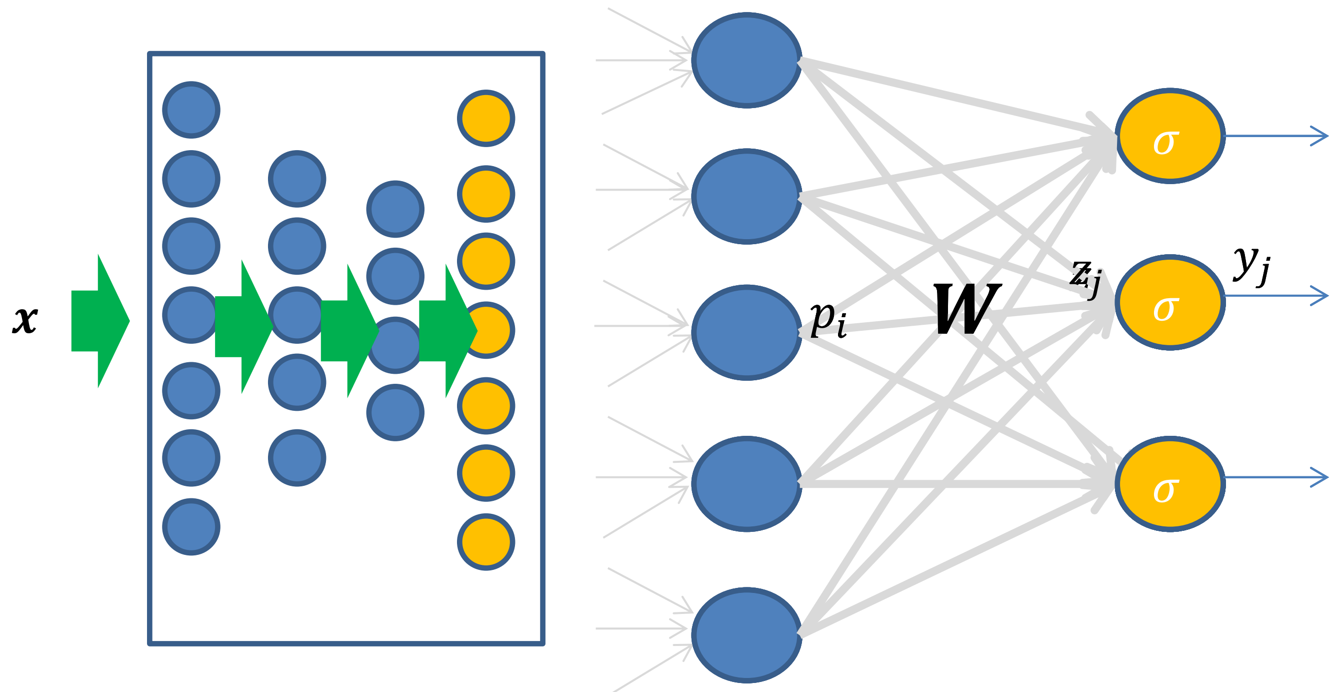
- When we want to use gradient descent for neural network learning, we need gradients over our loss

- For perceptrons, we could easily derive ∇F ourselves (by hand)

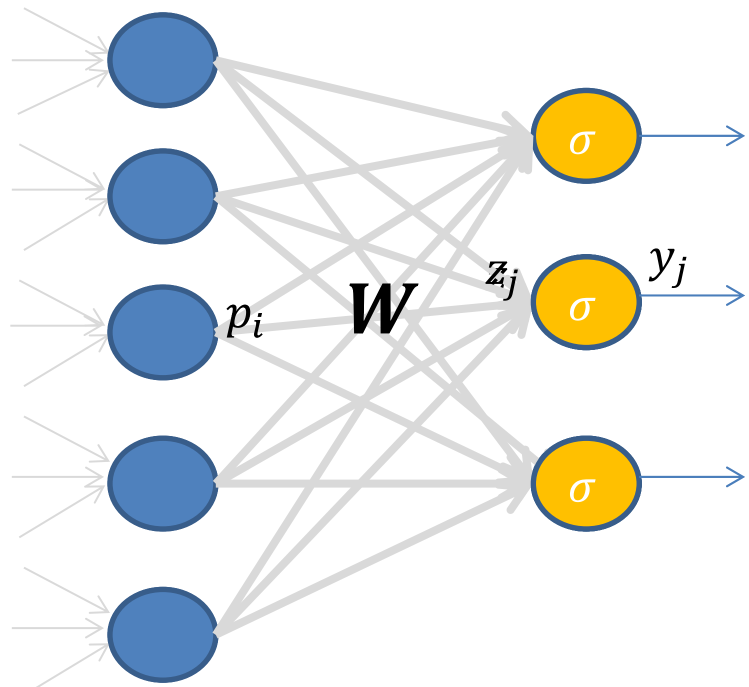
- For general MLP, we cannot derive ∇F so easily:
  - How to derive ∇F in these situations is the scope of the backprop algorithm
  - We write E (for error) instead of F throughout

- Our following mathematical derivations are based on
  - N. Buduma, Fundamentals of Deep Learning: Designing Next Generation Maching Intelligence Algorithms, Chapter 2
- If you enjoy another viewpoint have a look at
  - A. Karpathy, cs231n, 2016, Lecture 4, Backpropagation

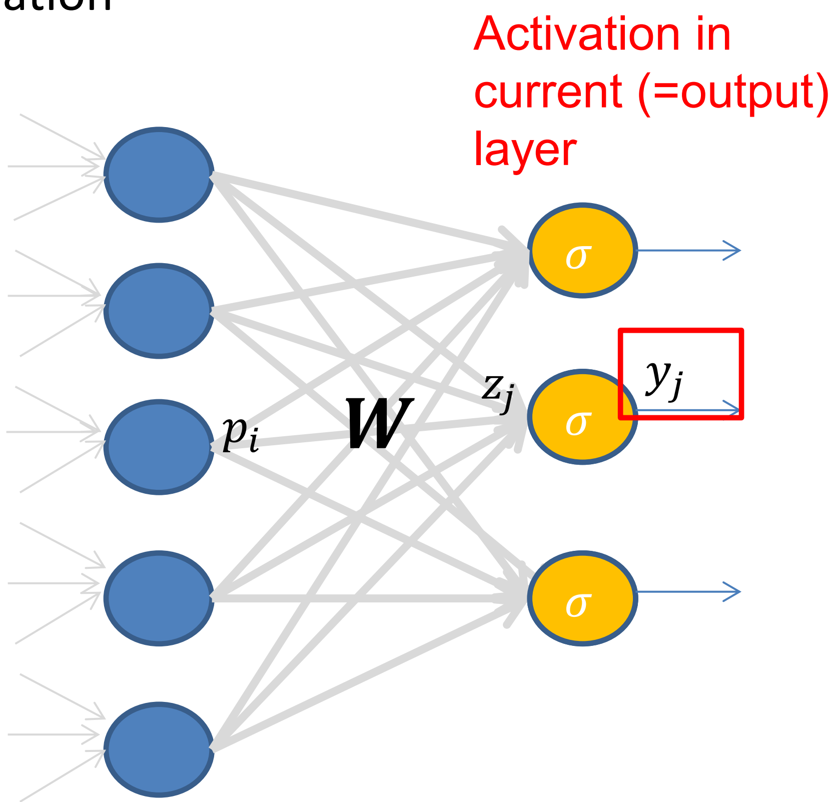
- Consider the following model situation



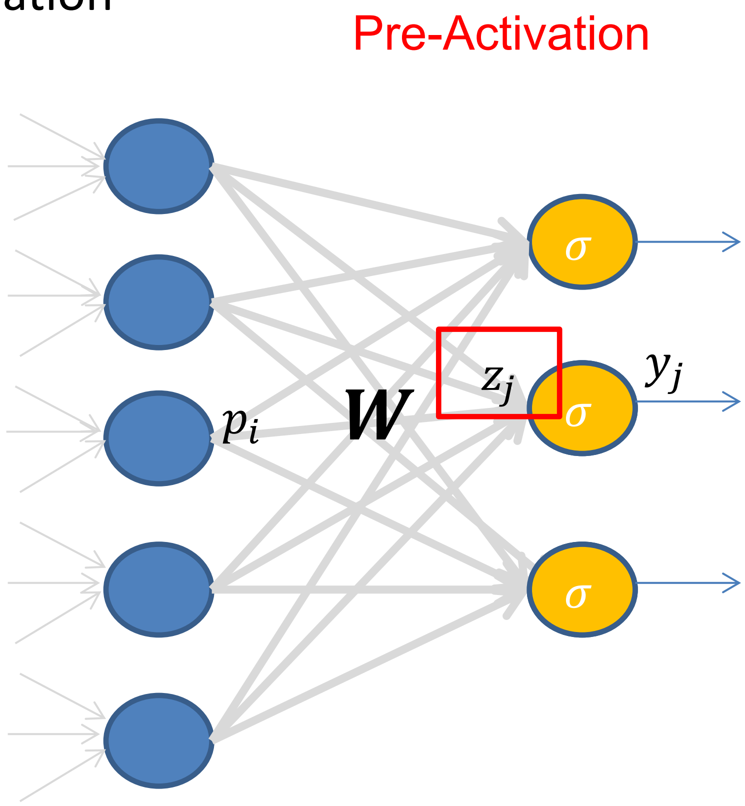
- Consider the following model situation



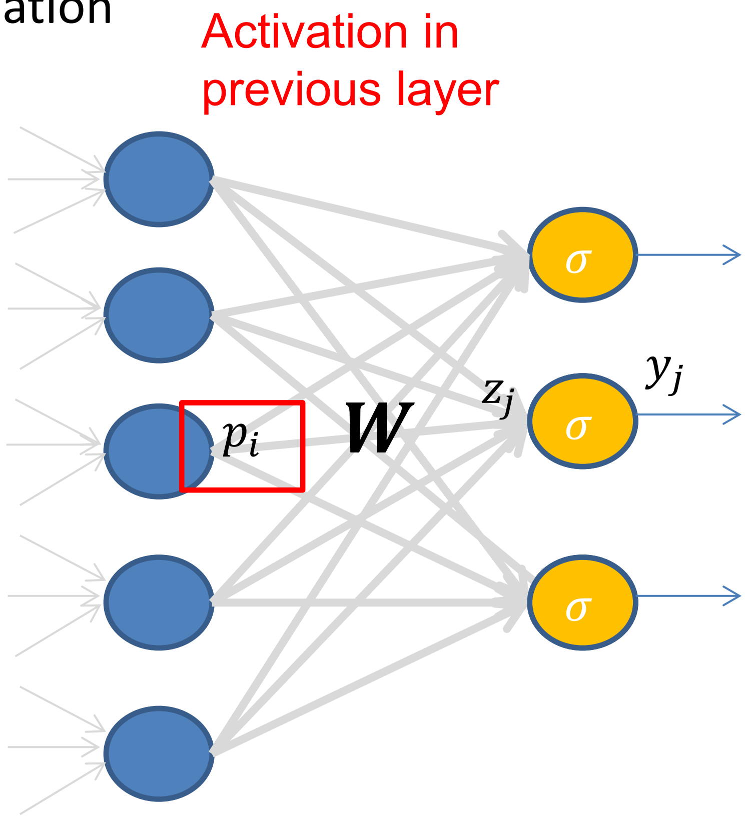
- Consider the following model situation



- Consider the following model situation



- Consider the following model situation



- Backpropagation has two phases:
  - 1. Forward Propagation
  - 2. Backward Propagation

- In 1. Forward propagation, all the (pre-)activations in all layers are computed
  - Starting from an input point $(\boldsymbol{x}, \boldsymbol{t})$

- We assume this has been done and focus on 2. Backward Propagation in the following

- Consider the above described model situation with some loss function
  - $E(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},\boldsymbol{t})} \ell(\boldsymbol{y}, \boldsymbol{t})$
  - The output $\boldsymbol{y} \in R^m$ is determined by some (deep) MLP

- We focus on minimizing
  - $E(\boldsymbol{\theta}) = \ell(\boldsymbol{y}, \boldsymbol{t})$
  - for notational convenience

- We focus on (multi-dim) square loss, but you can substitute other loss functions and derive analogous results
  - $\ell(\boldsymbol{y}, \boldsymbol{t}\,) = \left|\left|\boldsymbol{y} - \boldsymbol{t}\right|\right|^2$

- Consider the above described model situation with some loss function
  - $E(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},\boldsymbol{t})} \ell(\boldsymbol{y}, \boldsymbol{t})$
  - The output $\boldsymbol{y} \in R^m$ is determined by some (deep) MLP

- We focus on minimizing
  - $E(\boldsymbol{\theta}) = \ell(\boldsymbol{y}, \boldsymbol{t})$
  - for notational convenience

- We focus on (multi-dim) square loss, but you can substitute other loss functions and derive analogous results
  - $\ell(\boldsymbol{y}, \boldsymbol{t}\,) = \left|\left|\boldsymbol{y} - \boldsymbol{t}\right|\right|^2 = \sum_j \left(t_j - y_j\right)^2 = \sum_j e_j\left(y_j\right)$
    - Here, $e_j\left(y_j\right) = \left(t_j - y_j\right)^2$

High-level view of backprop

- Backprop is a form of dynamic programming
  - Recursively / inductively find the solution for an (optimization) problem

- We want to recursively determine the derivative of the loss wrt to the weights

- But we initially don't know how to do this, so we start out by looking at
  - $\delta \coloneqq \partial E / \partial q_i$
  - "how much does my loss/error change when activation in some neuron (in some layer) changes"
  - Then we relate δ to $\frac{\partial E}{\partial w_{ij}}$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is

$$\boxed{E = \sum_j e_j(y_j)}$$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is
- We find
  - $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial e_j}{\partial p_i}$

$$E = \sum_j e_j(y_j)$$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is
- We find
  - $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial p_i}$

$$\boxed{E = \sum_j e_j(y_j)}$$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is

Chain rule

- We find
  - $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial p_i}$

$$E = \sum_j e_j(y_j)$$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is
- We find
  - $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial p_i}$ (note that $\frac{\partial e_j}{\partial y_j} = \frac{\partial E}{\partial y_j}$)

$$\boxed{E = \sum_j e_j(y_j)}$$

$$\begin{aligned} E = e_1(y_1) \\ + e_2(y_2) \\ + \cdots \\ + e_j(y_j) \\ + \cdots \end{aligned}$$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is
- We find
  - $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial p_i}$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is
- We find
  - $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial p_i}$
- Now
  - $y_j = \sigma(z_j)$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is
- We find
  - $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial p_i}$
- Now
  - $y_j = \sigma(z_j)$
  - Therefore $\frac{\partial y_j}{\partial p_i} = \sigma'(z_j) \frac{\partial z_j}{\partial p_i}$

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is
- We find
  - $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial p_i}$
- Now
  - $y_j = \sigma(z_j)$
  - Therefore $\frac{\partial y_j}{\partial p_i} = \sigma'(z_j) \frac{\partial z_j}{\partial p_i}$
  - But:
    - $z_j = \boldsymbol{p} \cdot \boldsymbol{w}_j$
    - $\frac{\partial z_j}{\partial p_i} = [\boldsymbol{w}_j]_i$ (note that $[\boldsymbol{w}_j]_i = w_{ij}$)

- We start by asking ourselves what
  - $\frac{\partial E}{\partial p_i}$ is
- We find
  - $\boxed{\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial p_i}}$
- Now
  - $y_j = \sigma(z_j)$
  - Therefore $\boxed{\frac{\partial y_j}{\partial p_i} = \sigma'(z_j) \frac{\partial z_j}{\partial p_i}}$
  - But:
    - $z_j = \boldsymbol{p} \cdot \boldsymbol{w}_j$
    - $\boxed{\frac{\partial z_j}{\partial p_i} = w_{ij}}$
- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$ ⟵ We expressed error derivative in previous layer in terms of the current layer

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- That‘s a great result

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- That's a great result

- We already know what the error derivative wrt. the last layer is
  - $\frac{\partial E}{\partial y_j} = -2(t_j - y_j) = 2(y_j - t_j)$

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- That's a great result

- We already know what the error derivative wrt. the last layer is
  - $\frac{\partial E}{\partial y_j} = 2(y_j - t_j)$

- But from this we know the values at the layer (lastLayer-1) by our formula above

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- That's a great result

- We already know what the error derivative wrt. the last layer is
  - $\frac{\partial E}{\partial y_j} = 2(y_j - t_j)$

- But from this we know the values at the layer (lastLayer-1) by our formula above
- But from this we know the values at the layer (lastLayer-2) by our formula above
-

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- That's a great result

- We already know what the error derivative wrt. the last layer is
  - $\frac{\partial E}{\partial y_j} = 2(y_j - t_j)$

- But from this we know the values at the layer (lastLayer-1) by our formula above
- But from this we know the values at the layer (lastLayer-2) by our formula above
- ……………………………

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- That‘s a great result

- We already know what the error derivative wrt. the last layer is
  - $\frac{\partial E}{\partial y_j} = 2(y_j - t_j)$

- But from this we know the values at the layer (lastLayer-1) by our formula above
- But from this we know the values at the layer (lastLayer-2) by our formula above
- …………………………
- We **backpropagate** the error derivatives from the last layer to the very first!

But we‘re looking for derivatives wrt. weights!

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- But we‘re looking for
  - $\frac{\partial E}{\partial u_{ik}}$ for all of the weight matrices $\boldsymbol{U} = \boldsymbol{W}^{(\mathbf{1})}, \boldsymbol{W}^{(\mathbf{2})}, \boldsymbol{W}^{(\mathbf{3})}, \ldots$

But we‘re looking for derivatives wrt. weights!

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

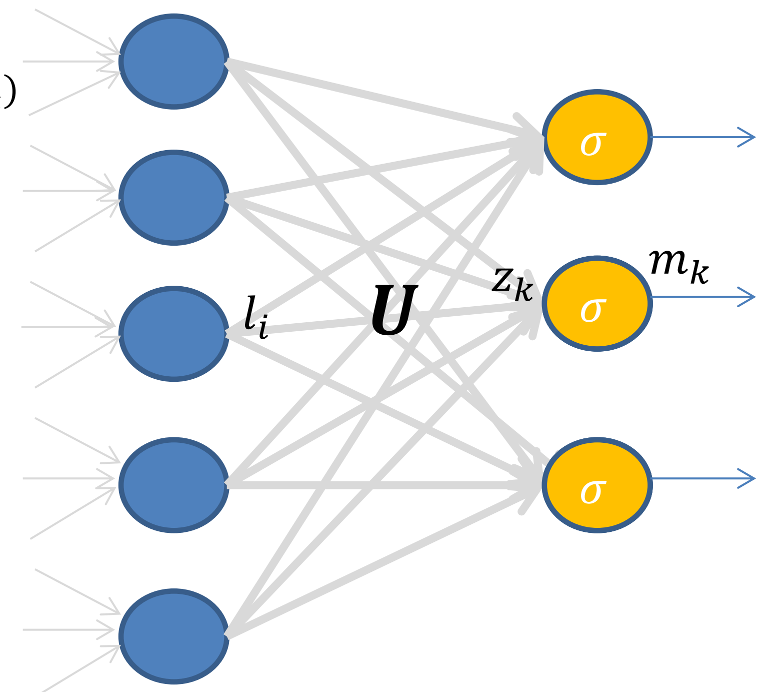
We expressed error derivative in previous layer in terms of the current layer

- But we‘re looking for
  - $\frac{\partial E}{\partial u_{ik}}$ for all of the weight matrices $\boldsymbol{U} = \boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{W}^{(3)}, \ldots$

  - Fortunately, we have the relation
    - $\frac{\partial E}{\partial u_{ik}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial u_{ik}} = \frac{\partial E}{\partial m_k} \frac{\partial m_k}{\partial z_k} l_i = \boxed{\frac{\partial E}{\partial m_k}} \sigma'(z_k) l_i$
      - Here, m_k is the output/activation at the layer corresponding to matrix $\boldsymbol{U}$
      - $\boldsymbol{l}$ is the input for that layer
      - z_k is the pre-activation of m_k, i.e., $m_k = \sigma(z_k)$

But we‘re looking for derivatives wrt. weights!

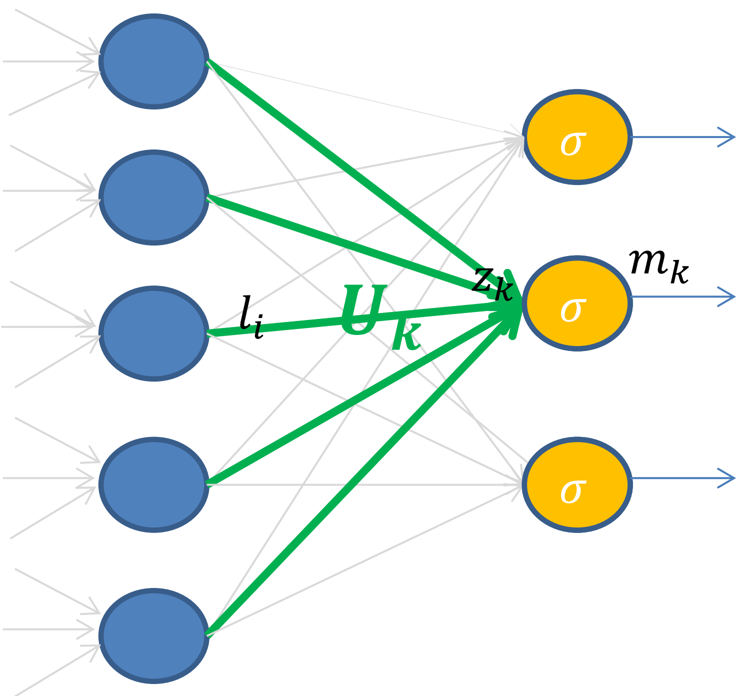
- Fortunately, we have the relation
  - $\frac{\partial E}{\partial u_{ik}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_{ik}} = \frac{\partial E}{\partial m_k}\frac{\partial m_k}{\partial z_k} l_i = \frac{\partial E}{\partial m_k}\sigma'(z_k) l_i$
    - Here, m_k is the output/activation at the layer corresponding to matrix $\boldsymbol{U}$
    - $\boldsymbol{l}$ is the input for that layer
    - z_k is the pre-activation of m_k, i.e., $m_k = \sigma(z_k)$



But we‘re looking for derivatives wrt. weights!

$\boldsymbol{U} = [\boldsymbol{U}_1 \cdots \boldsymbol{U}_k \cdots]$

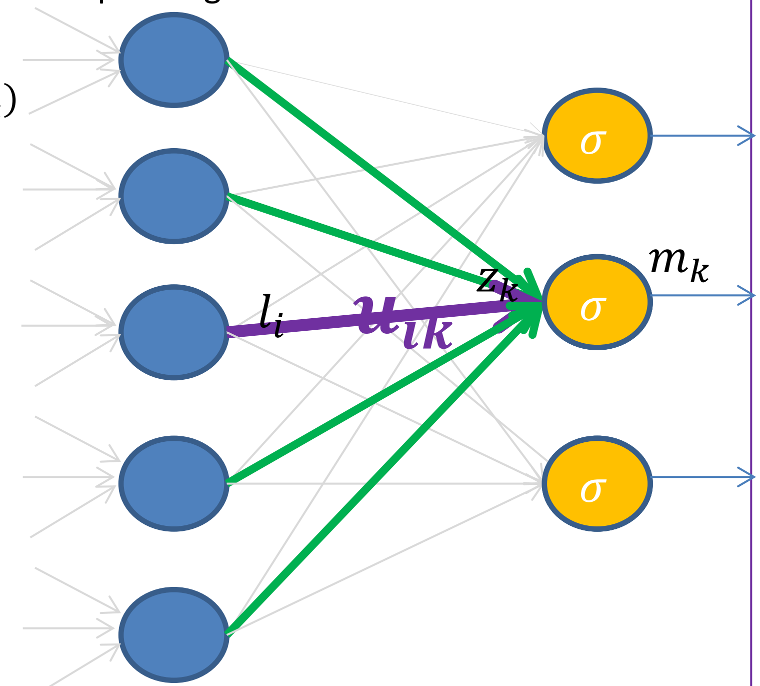
- Fortunately, we have the relation
  - $\frac{\partial E}{\partial u_{ik}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_{ik}} = \frac{\partial E}{\partial m_k}\frac{\partial m_k}{\partial z_k} l_i = \frac{\partial E}{\partial m_k}\sigma'(z_k) l_i$
    - Here, m_k is the output/activation at the layer corresponding to matrix $\boldsymbol{U}$
    - $\boldsymbol{l}$ is the input for that layer
    - z_k is the pre-activation of m_k, i.e., $m_k = \sigma(z_k)$



But we‘re looking for derivatives wrt. weights!

$$\boldsymbol{U} = [\boldsymbol{U}_1 \cdots \boldsymbol{U}_k \cdots]$$

- Fortunately, we have the relation
  - $$\frac{\partial E}{\partial u_{ik}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_{ik}} = \frac{\partial E}{\partial m_k}\frac{\partial m_k}{\partial z_k} l_i = \frac{\partial E}{\partial m_k}\sigma'(z_k) l_i$$
    - Here, m_k is the output/activation at the layer corresponding to matrix $\boldsymbol{U}$
    - $\boldsymbol{l}$ is the input for that layer
    - z_k is the pre-activation of m_k, i.e., $m_k = \sigma(z_k)$

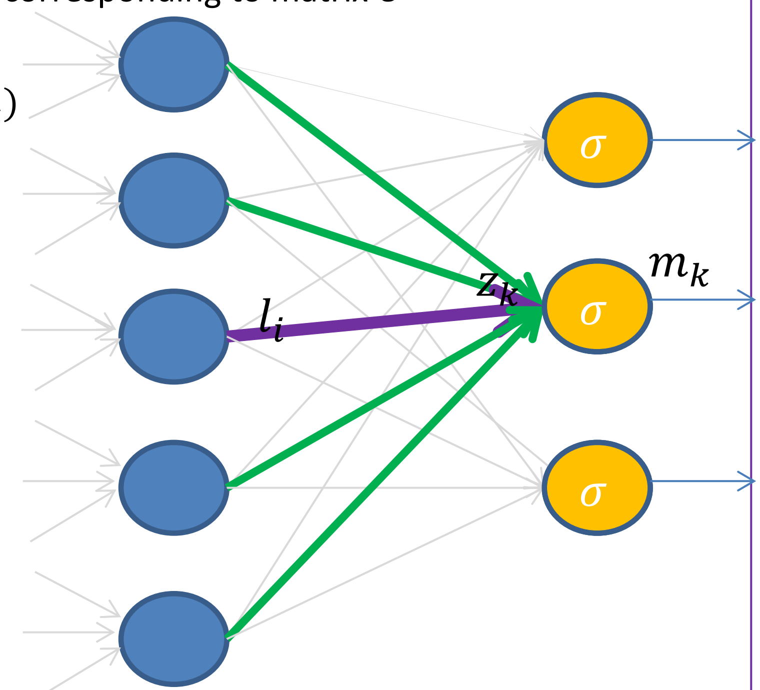


But we‘re looking for derivatives wrt. weights!

Chain rule

$$\boldsymbol{U} = [\boldsymbol{U}_1 \cdots \boldsymbol{U}_k \cdots]$$

- Fortunately, we have the relation
  - $$\frac{\partial E}{\partial u_{ik}} = \boxed{\frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_{ik}}} = \frac{\partial E}{\partial m_k}\frac{\partial m_k}{\partial z_k} l_i = \frac{\partial E}{\partial m_k}\sigma'(z_k) l_i$$
    - Here, m_k is the output/activation at the layer corresponding to matrix $\boldsymbol{U}$
    - $\boldsymbol{l}$ is the input for that layer
    - z_k is the pre-activation of m_k, i.e., $m_k = \sigma(z_k)$

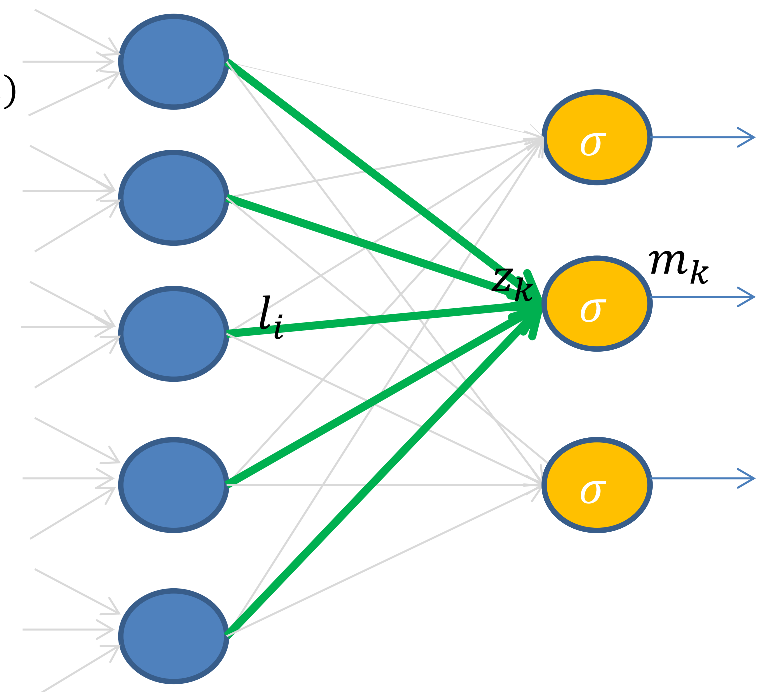


But we‘re looking for derivatives wrt. weights!

$$\boldsymbol{U} = [\boldsymbol{U}_1 \cdots \boldsymbol{U}_k \cdots]$$

- Fortunately, we have the relation
  - $\frac{\partial E}{\partial u_{ik}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_{ik}} = \frac{\partial E}{\partial m_k}\frac{\partial m_k}{\partial z_k}\boxed{l_i} = \frac{\partial E}{\partial m_k}\sigma'(z_k)l_i$
    - Here, m_k is the output/activation at the layer corresponding to matrix $\boldsymbol{U}$
    - $\boldsymbol{l}$ is the input for that layer
    - z_k is the pre-activation of m_k, i.e., $m_k = \sigma(z_k)$

$$z_k = \boldsymbol{l} \cdot \boldsymbol{U}_k$$
$$= \sum_i l_i (U_k)_i$$
$$= \sum_i l_i u_{ik}$$



But we‘re looking for derivatives wrt. weights!

TECHNISCHE UNIVERSITÄT DARMSTADT

- $\frac{\partial E}{\partial p_i} = \sum_j \frac{\partial E}{\partial y_j} \sigma'(z_j) w_{ij}$

We expressed error derivative in previous layer in terms of the current layer

- But we‘re looking for
  - $\frac{\partial E}{\partial u_{ik}}$ for all of the weight matrices $\boldsymbol{U} = \boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{W}^{(3)}, \ldots$
  - Fortunately, we have the relation

This we get from backprop

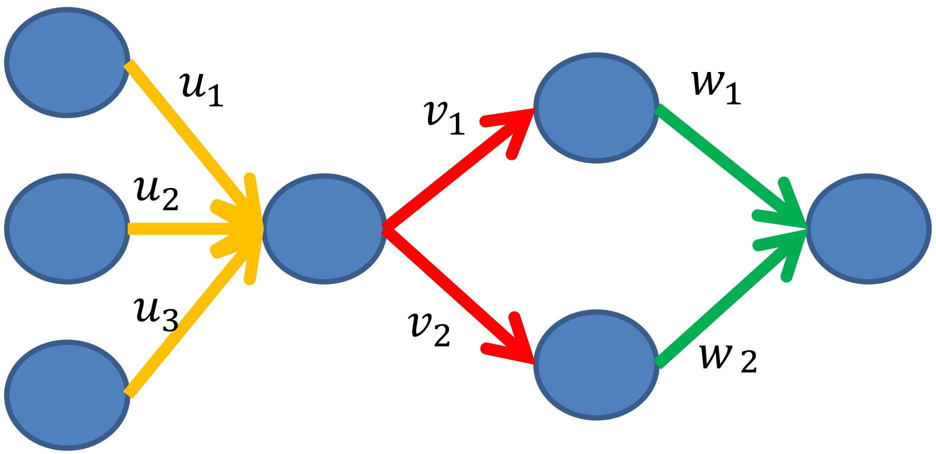
- $\frac{\partial E}{\partial u_{ik}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial u_{ik}} = \frac{\partial E}{\partial m_k} \frac{\partial m_k}{\partial z_k} l_i = \boxed{\frac{\partial E}{\partial m_k}} \sigma'(z_k) l_i$
  - Here, m_k is the output/activation at the layer corresponding to matrix $\boldsymbol{U}$
  - $\boldsymbol{l}$ is the input for that layer
  - z_k is the pre-activation of m_k, i.e., $m_k = \sigma(z_k)$

Summary

- Backpropagation is a recursive algorithm for determining error derivatives
- Starts at the outer layer
- Propagates error derivative signal 'backwards'
  - By expressing derivatives in one layer in terms of the next layer derivatives using the chain rule
- Note that it may be a general technique for calculating derivatives of composite functions
  - Modifcations of the presented algorithm apply to other mathematical functions, too (not only neural nets!)

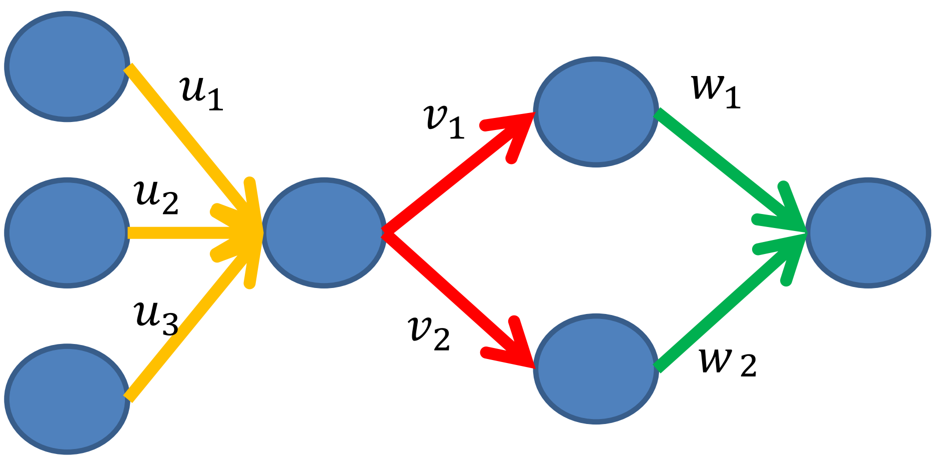
Example

- We look at the following net
- Assume all activation functions are tanh, loss is square loss



Example

- We look at the following net
- Assume all activation functions are tanh, loss is square loss
- Let's initialize our net to $\boldsymbol{u} = (0.2, 0.5, -1), \boldsymbol{v} = (0.9, -0.5), \boldsymbol{w} = (0.2, -5)$

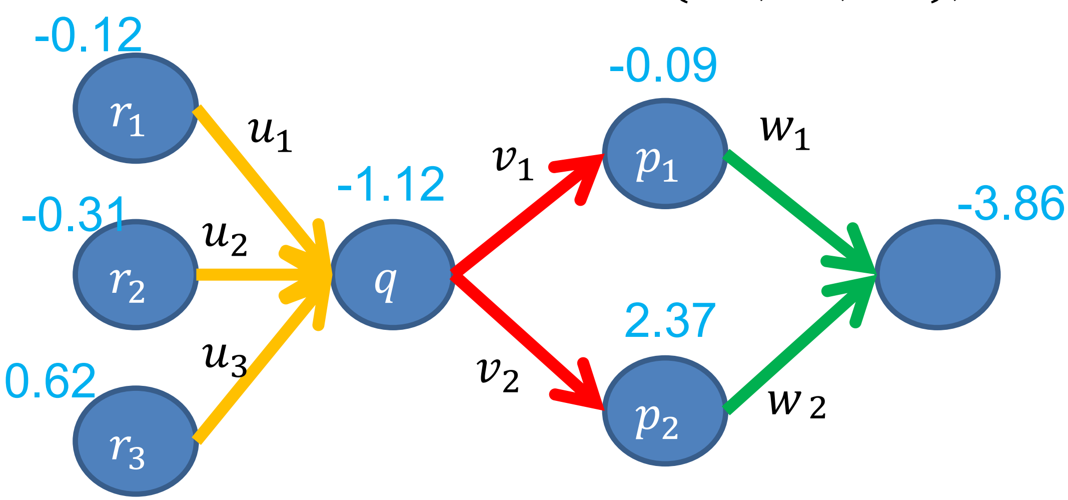


- Assume that $\boldsymbol{x} = (1,0,1)$ and $t = 1$
- Square loss: $(t-y)^2$

- We first perform a **forward pass** and compute everything – from activations to loss. Then we switch to the **backward pass**

Example

- We look at the following net
- Assume all activation functions are tanh, loss is square loss
- Let's initialize our net to $\boldsymbol{u} = (0.2, 0.5, -1), \boldsymbol{v} = (0.9, -0.5), \boldsymbol{w} = (0.2, -5)$



- Assume that $\boldsymbol{x} = (1,0,1)$ and $t = 1$
- Square loss: $(t - y)^2$

$$\boxed{\frac{\partial E}{\partial u_{ik}} = \frac{\partial E}{\partial m_k} \sigma'(z_k) l_i}$$

$$\frac{\partial E}{\partial u_3} = -1.12 \cdot (0.55) \cdot 1 = -0.61$$

Example – gradient check

TECHNISCHE UNIVERSITÄT DARMSTADT

- Finally, to see if we did everything correctly, we (can) perform a **numeric gradient check**
- E.g. to check $\partial E / \partial u_3$

Recall: $f'(x) \approx \frac{1}{h}(f(x+h) - f(x))$

  - We compute our loss E at (for $\boldsymbol{x} = (1{,}0{,}1)$ and $t = 1$)

$$\boldsymbol{u}' = (0.2{,}0.5, -1 + h), \boldsymbol{v} = (0.9, -0.5), \boldsymbol{w} = (0.2, -5)$$

  - and at $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$

  - Then, we compute

$$\frac{1}{h} \cdot \left(E(\boldsymbol{x}, t;\ \boldsymbol{u}', \boldsymbol{v}, \boldsymbol{w}) - E(\boldsymbol{x}, t;\ \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \right)$$

Outline

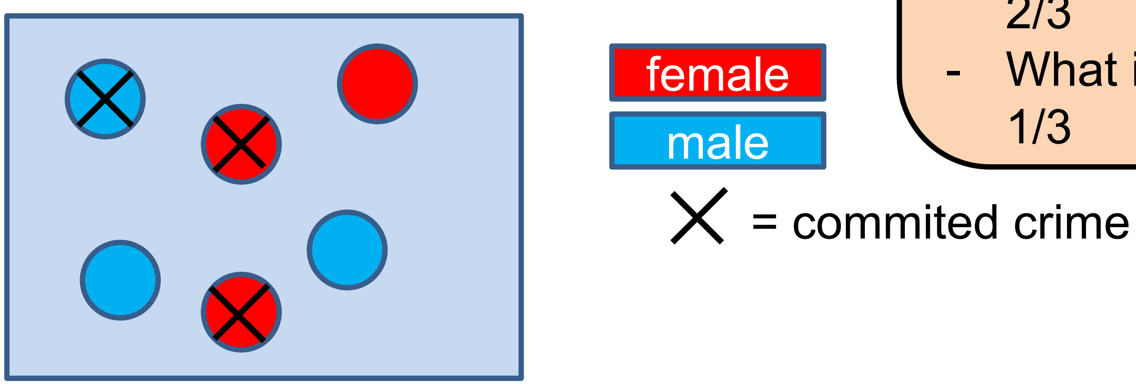
(Neural) Language Models

Language Model

- Language model:
  - Assigns a sequence of tokens (words, characters, …) a probability
    - Which denotes likelihood of observing this sequence in text
- Intuitively
  - Probability(*the cat sat on the mat*) > Probability(*mat sat cat on the the*)
- Use cases:
  - Scoring sequences: e.g. in MT
  - Generating Text

One slide primer on probability

- Joint probability: $P(A \cap B)$ also denoted as: $P(A, B)$, $P(A\ B)$
  - how likely is it to observe events A and B jointly?
- Conditional probability: $P(A|B)$
  - how likely is it to observe A given B?
- Marginal probability: $P(A)$
  - how likely is it to observe event A?



- What is P(female)? 1/2
- What is P(female,crime)? 2/6
- What is P(female|crime)? 2/3
- What is P(crime|female)? 2/3
- What is P(crime|male)? 1/3

N-gram language models

TECHNISCHE UNIVERSITÄT DARMSTADT

- In former times, a common approach was to use **n-gram** language models

- Approximate the true probability P of a stream of tokens
  - $P(w_1\ w_2\ w_3\ w_4\ \ldots.) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1 w_2) \cdot P(w_4|\ w_1 w_2 w_3) \cdot P(w_5|w_1 w_2 w_3 w_4) \cdots$

  - by an n-gram model, where:
    - $P(w_t|w_1 \cdots w_{t-1}) \approx P(w_t|w_{t-n+1} \cdots w_{t-1})$

NLLG

N-gram language models

- For example, 1-gram model (unigram)

$$P(w_1 w_2 w_3 w_4 \cdots) \approx P(w_1)P(w_2)P(w_3)P(w_4)\cdots$$

- 2-gram model (bigram)

$$P(w_1 w_2 w_3 w_4 \cdots) \approx P(w_1)P(w_2|w_1)P(w_3|w_2)P(w_4|w_3)\cdots$$

N-gram language models

TECHNISCHE UNIVERSITÄT DARMSTADT

- In traditional approaches, n-gram language probabilities are estimated from **counts** in (training) data:

$$P(\text{dog}|\text{the}) = \frac{\text{cnt(the,dog)}}{\text{cnt(the)}}$$

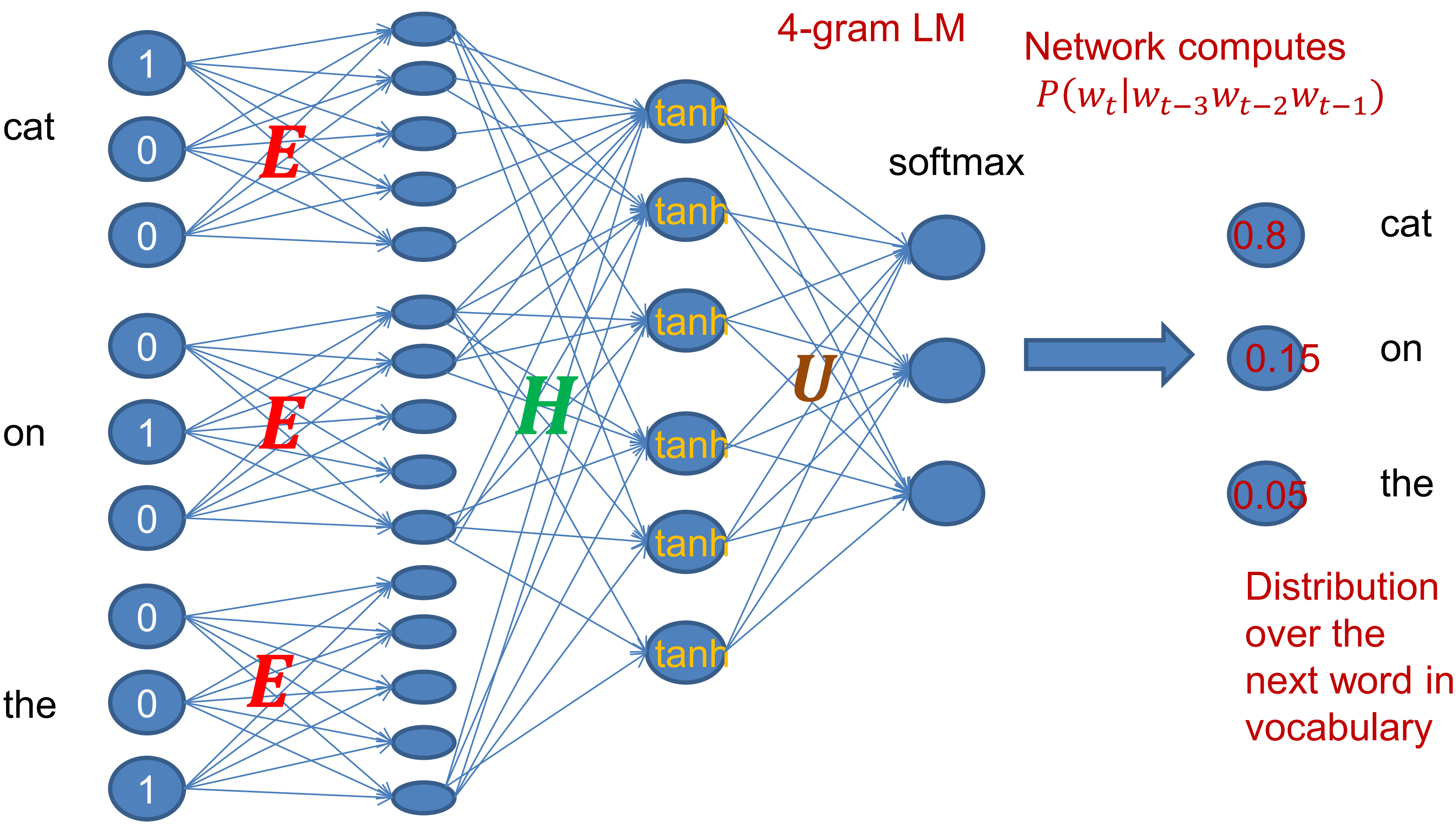
  - $P(w_t|w_{t-n+1}\cdots w_{t-1}) \approx \frac{\text{count}(w_{t-n+1}\cdots w_{t-1}w_t)}{\text{count}(w_{t-n+1}\cdots w_{t-1})}$

  - Additionally, some *smoothing* is performed to account for words not seen in the data
    - I.e. we should not assign zero probability to "the dog"
    - Just because we never saw this sequence in our training data (especially when we saw a dog and the cat etc.)

- Implementing an n-gram language model with an MLP is (also) easy …

NLLG

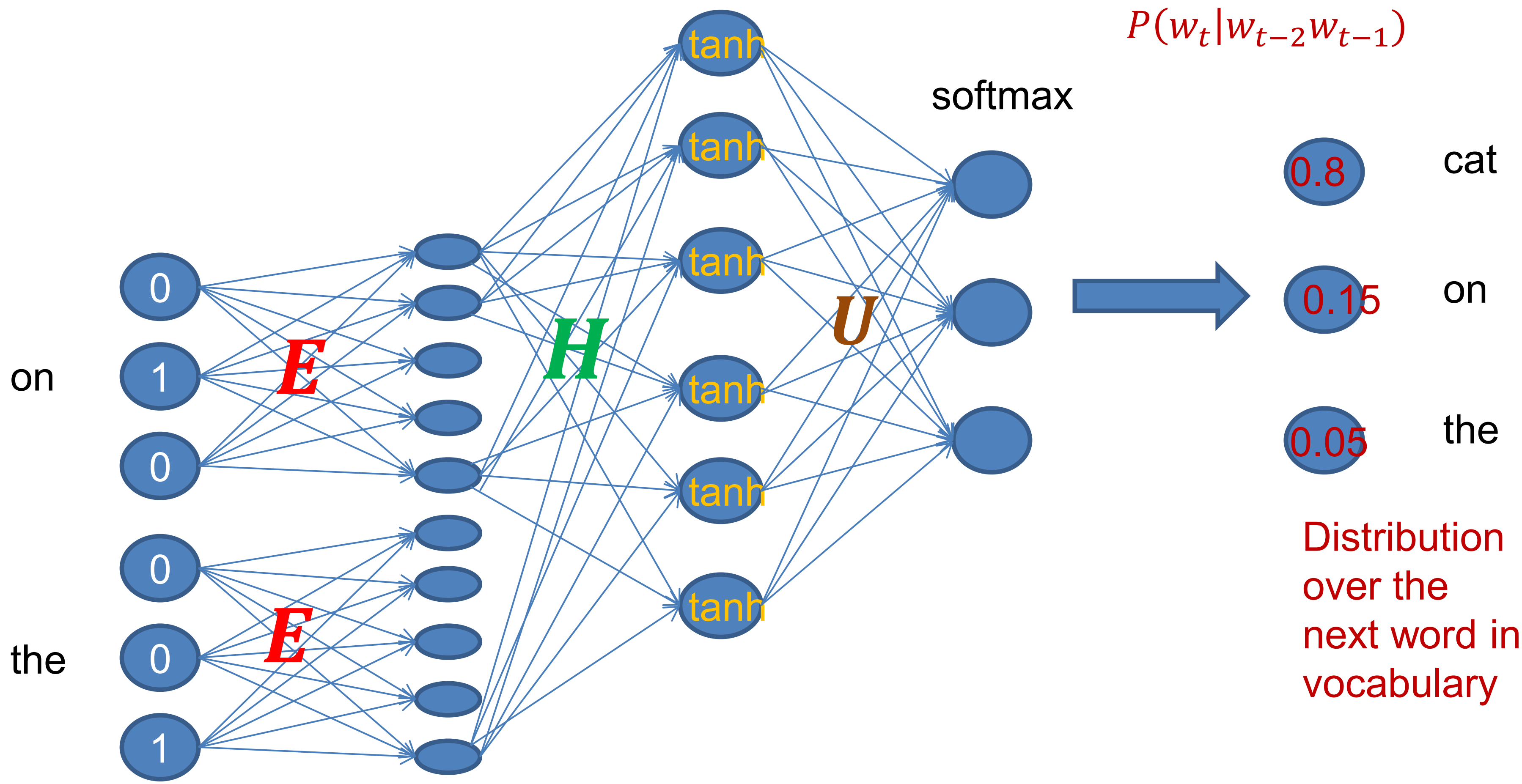
Bengio et al. (2003), A neural prob. language model



Bengio et al. (2003), A neural prob. language model

3-gram LM

Network computes $P(w_t|w_{t-2}w_{t-1})$



Distribution over the next word in vocabulary

Language Models can (also) be used to Generate Language

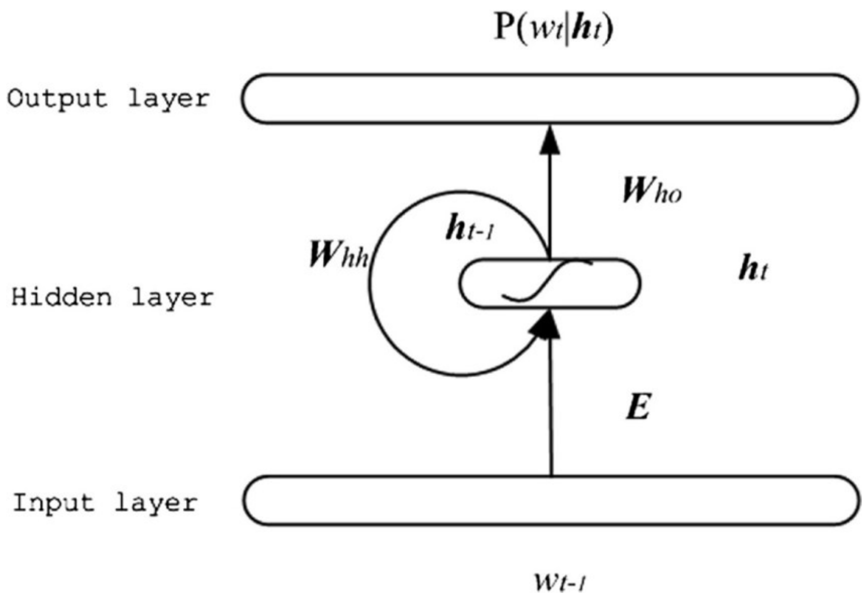
- For example with a bigram language model:
  - Sample $\widetilde{w}_1$ from $p(w_1)$
  - Then sample $\widetilde{w}_2$ from $p(w_2|\widetilde{w}_1)$
  - Then sample $\widetilde{w}_3$ from $p(w_3|\widetilde{w}_2)$
  - And so on

- The problem with n-gram language models is that they are inherently limited in the past window that they can take into consideration

RNN language models

TECHNISCHE UNIVERSITÄT DARMSTADT

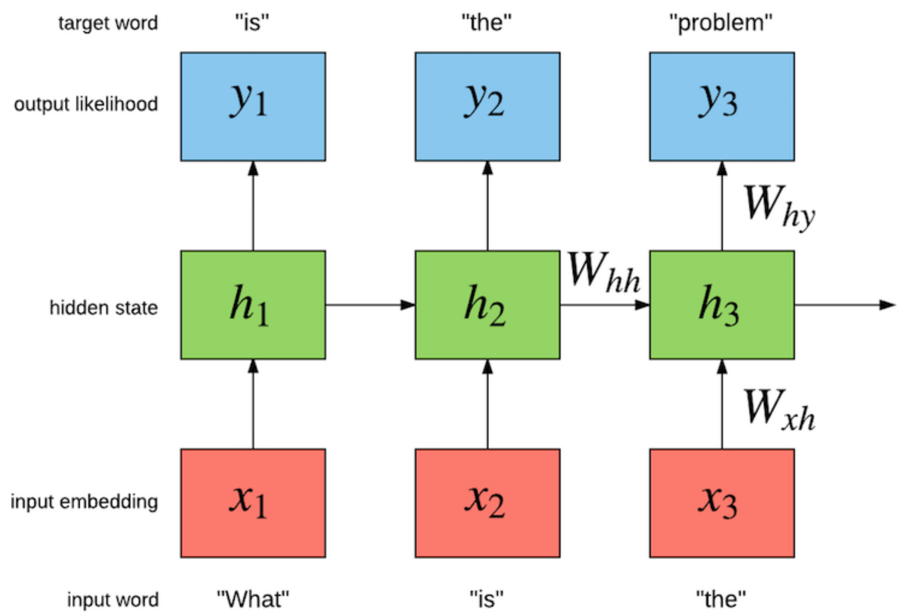
- To remedy, one uses a slightly different network structure, so called **Recurrent Neural Networks**
- Where indefinite amount of past knowledge is stored in the hidden layer



NLLG

RNN language models

- The RNN is trained to predict the next word

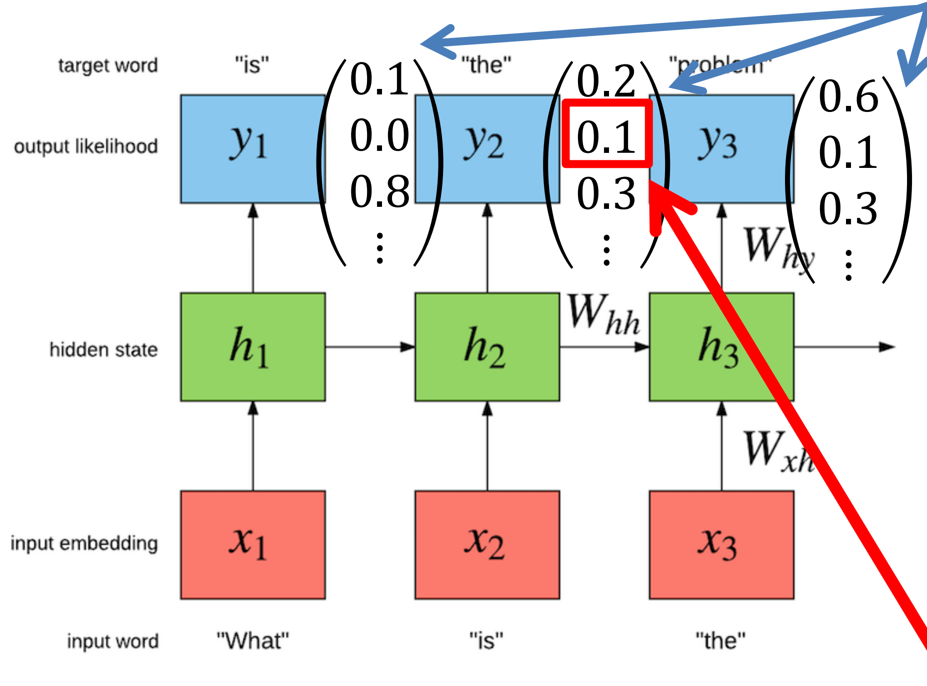


- The RNN computes $p(w_t|w_1 \cdots w_{t-1})$ → no n-gram approximation
- At generation time, sample from the softmax, and feed in to next time step

RNN language models

- Example

A probability distribution over the whole vocabulary at each time step



- To get $p(\text{the}|\text{What is})$, choose corresponding probability from softmax (e.g. 0.1)
- It is common to pad the input with a SOS and predict an EOS in the last step

Language Models in NLP

- LMs have seen a revival in NLP
  - E.g., because language modeling can be a cheap **auxiliary task** (see work of Marek Rei)
  - And also due to ELMo, a word embedding model that computes contextualized word embeddings using a language modeling objective

Summary

- We saw gradient descent (GD, SGD)
  - A general technique for optimization
  - Saw also extensions of vanilla SGD
- Then we saw backprop(agation)
  - A general technique for deriving gradients in MLPs
  - Once the gradient is determined, can again learn with GD or extensions
- Then we saw N-gram and RNN language models

References

- Senior, Andrew, Georg Heigold, and Ke Yang. "An empirical study of learning rates in deep neural networks for speech recognition." *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013.
- Bengio, Yoshua, et al. "A neural probabilistic language model." *Journal of machine learning research* 3.Feb (2003): 1137-1155.
- Sutskever, Ilya, James Martens, and Geoffrey E. Hinton. "Generating text with recurrent neural networks." *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011.
- Rei, Marek, Semi-supervised Multitask Learning for Sequence Labeling, ACL 2017