

Deep Learning for NLP

Universität Bielefeld

Lecture 3 – Learning in MLPs / Backprop

Dr. Steffen Eger

seminar.dldh@gmail.com



This lecture:

- Gradient Descent Learning:
 - A general optimization technique for non-linear functions
- Backpropagation
 - A general technique to determine the gradients in neural networks
- Language Modeling
 - A general approach to model natural languages

Gradient Descent

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 $F : \mathbb{R}^n \rightarrow \mathbb{R}$.

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

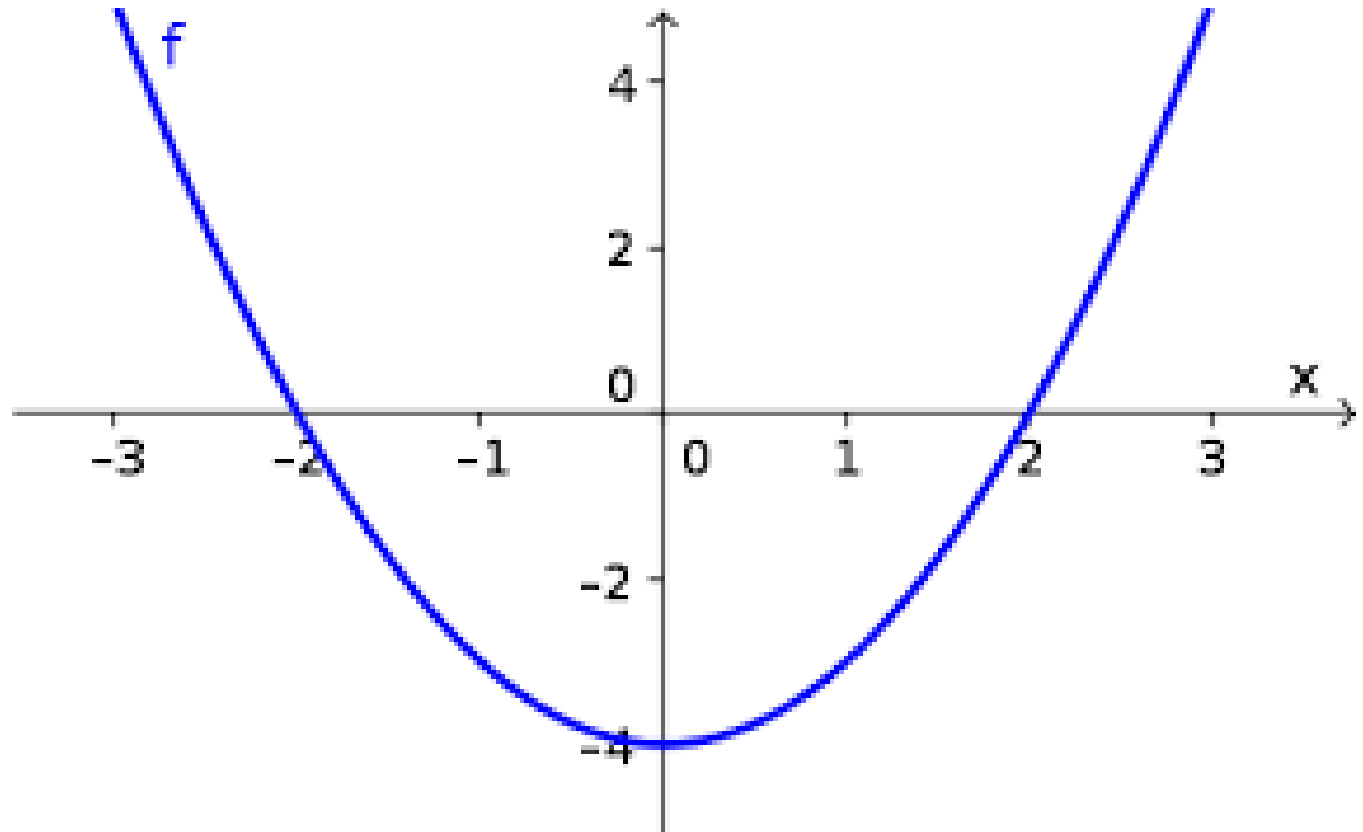
$$\mathbf{w}' \leftarrow \mathbf{w} - \alpha \nabla F(\mathbf{w})$$

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

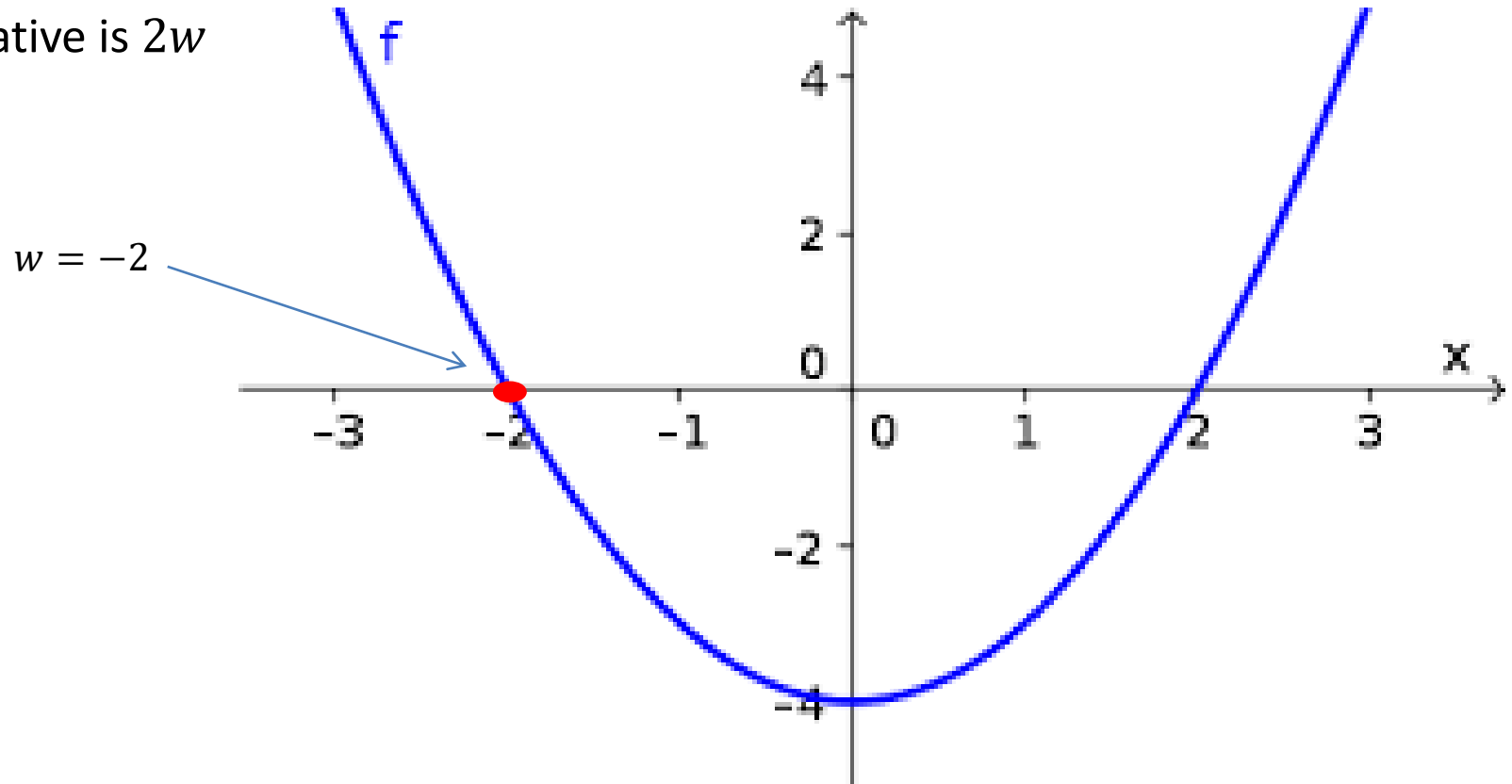
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



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

Gradient descent by example

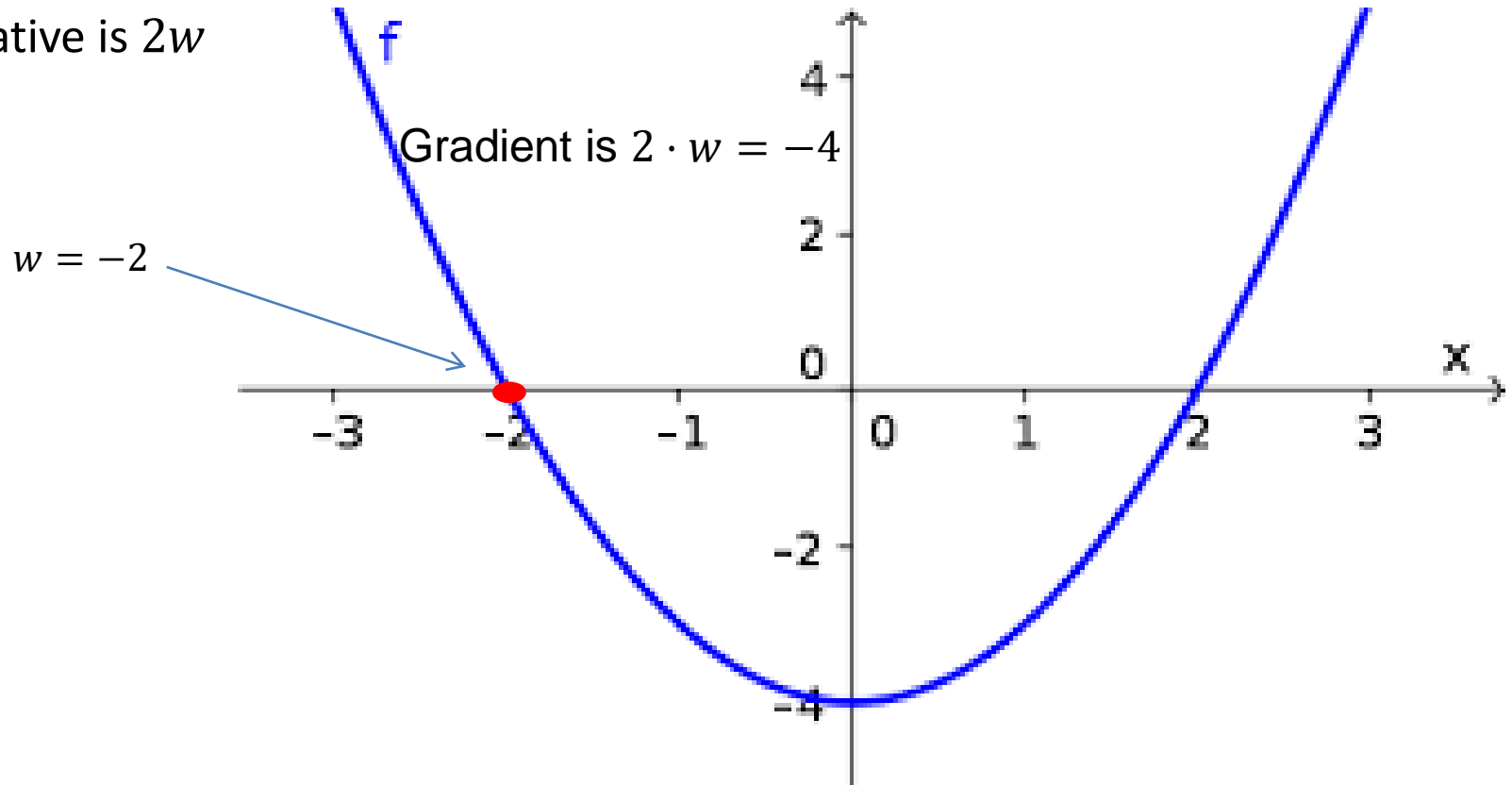
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



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

Gradient descent by example

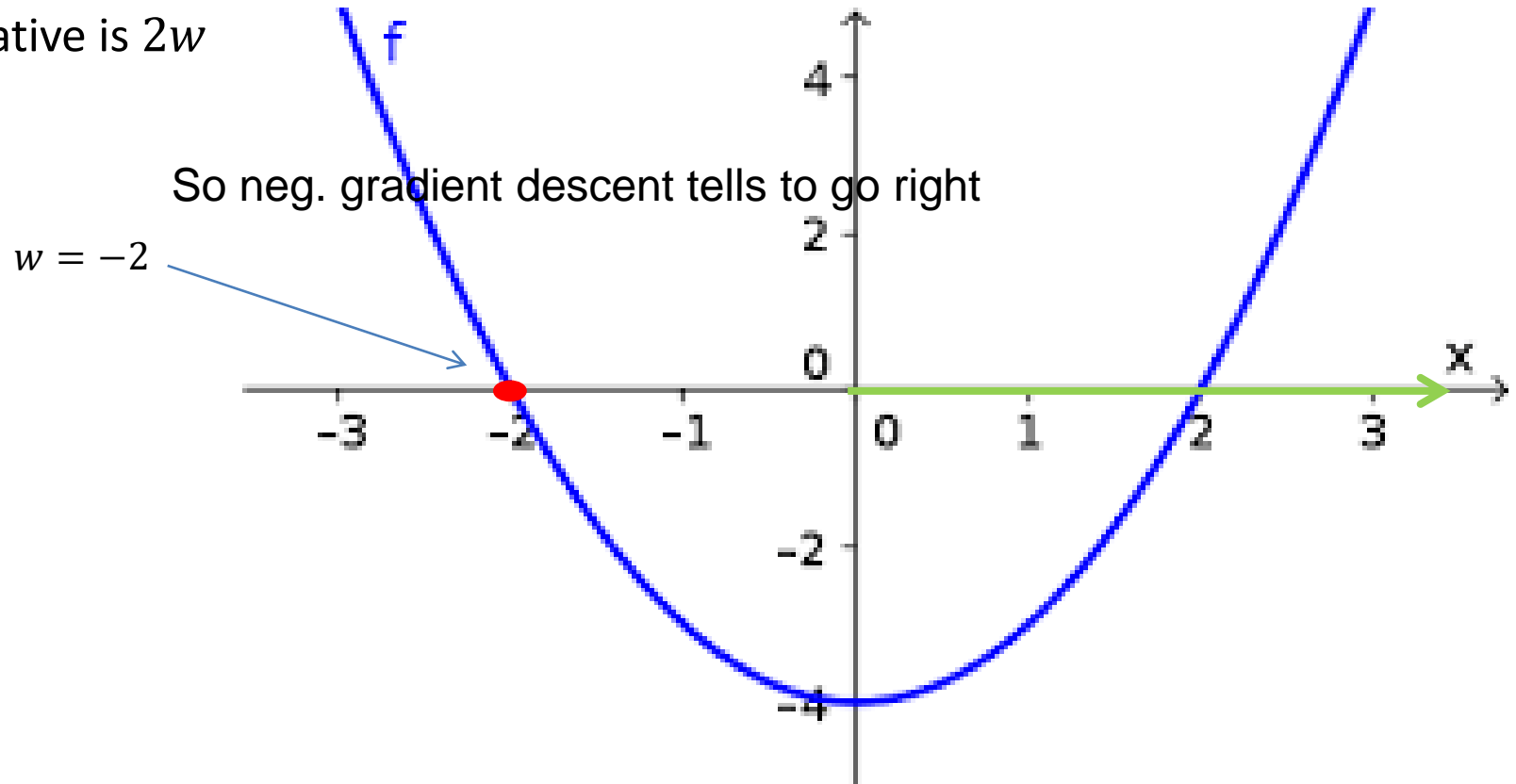
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



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

Gradient descent by example

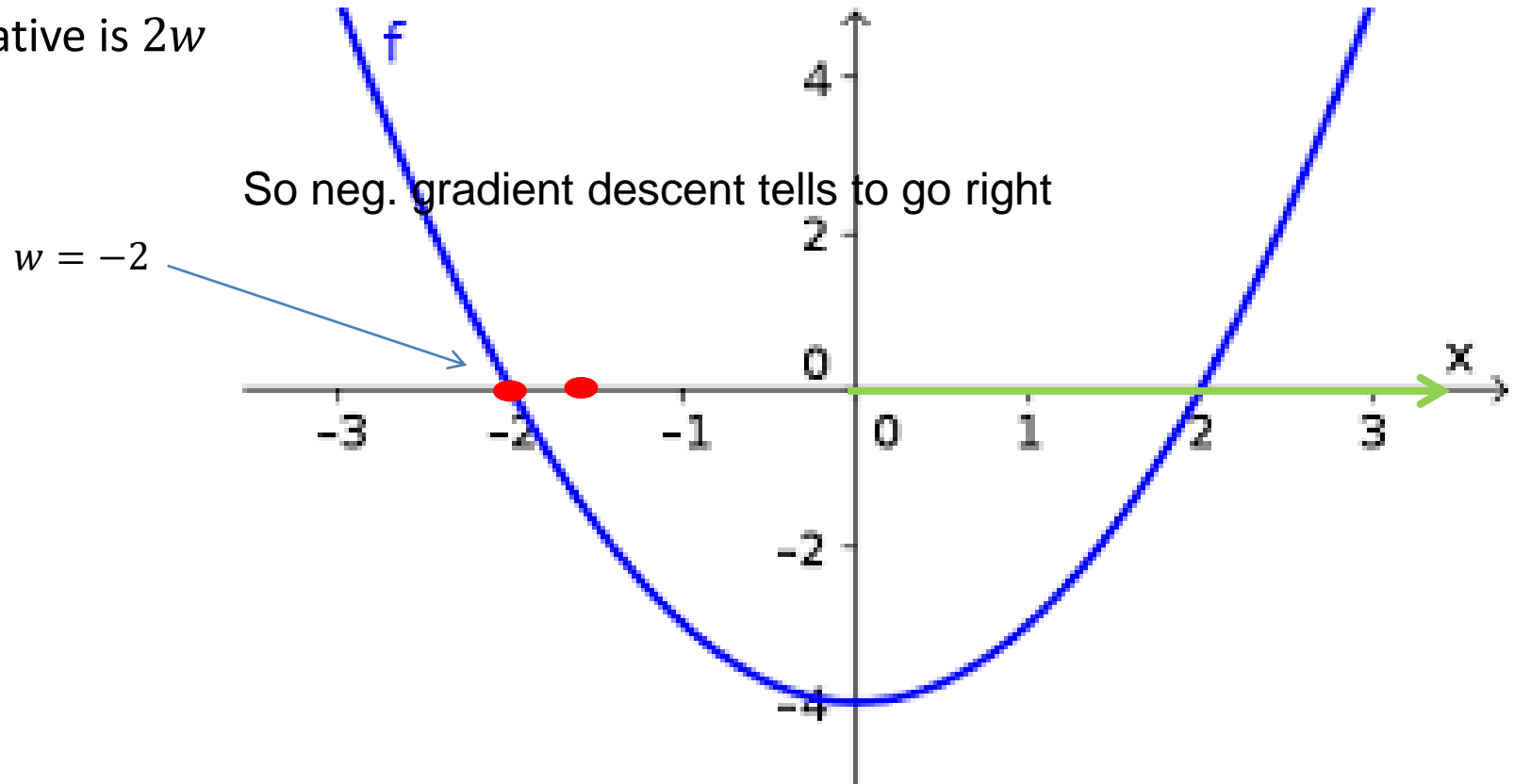
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



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

Gradient descent by example

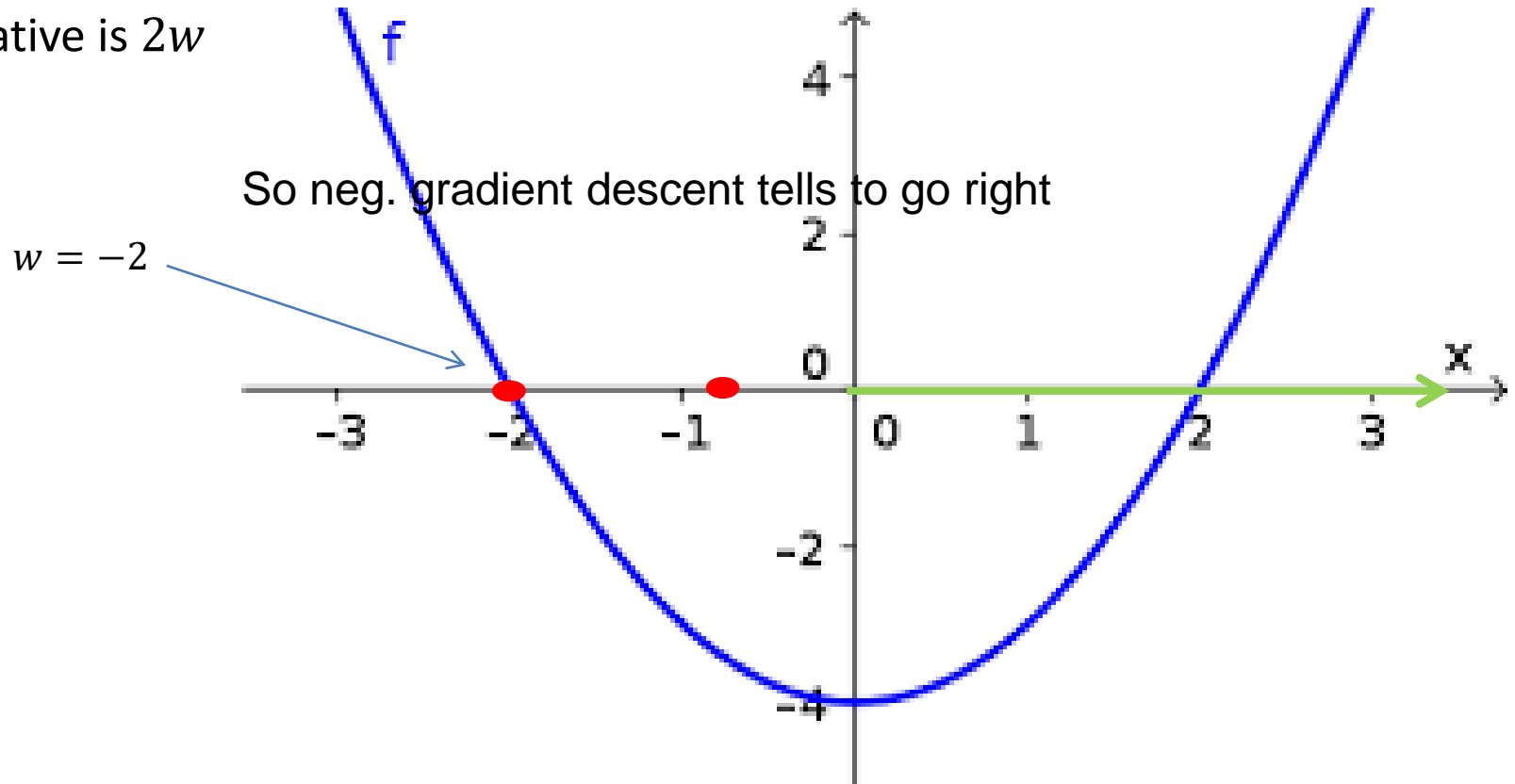
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



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

Gradient descent by example

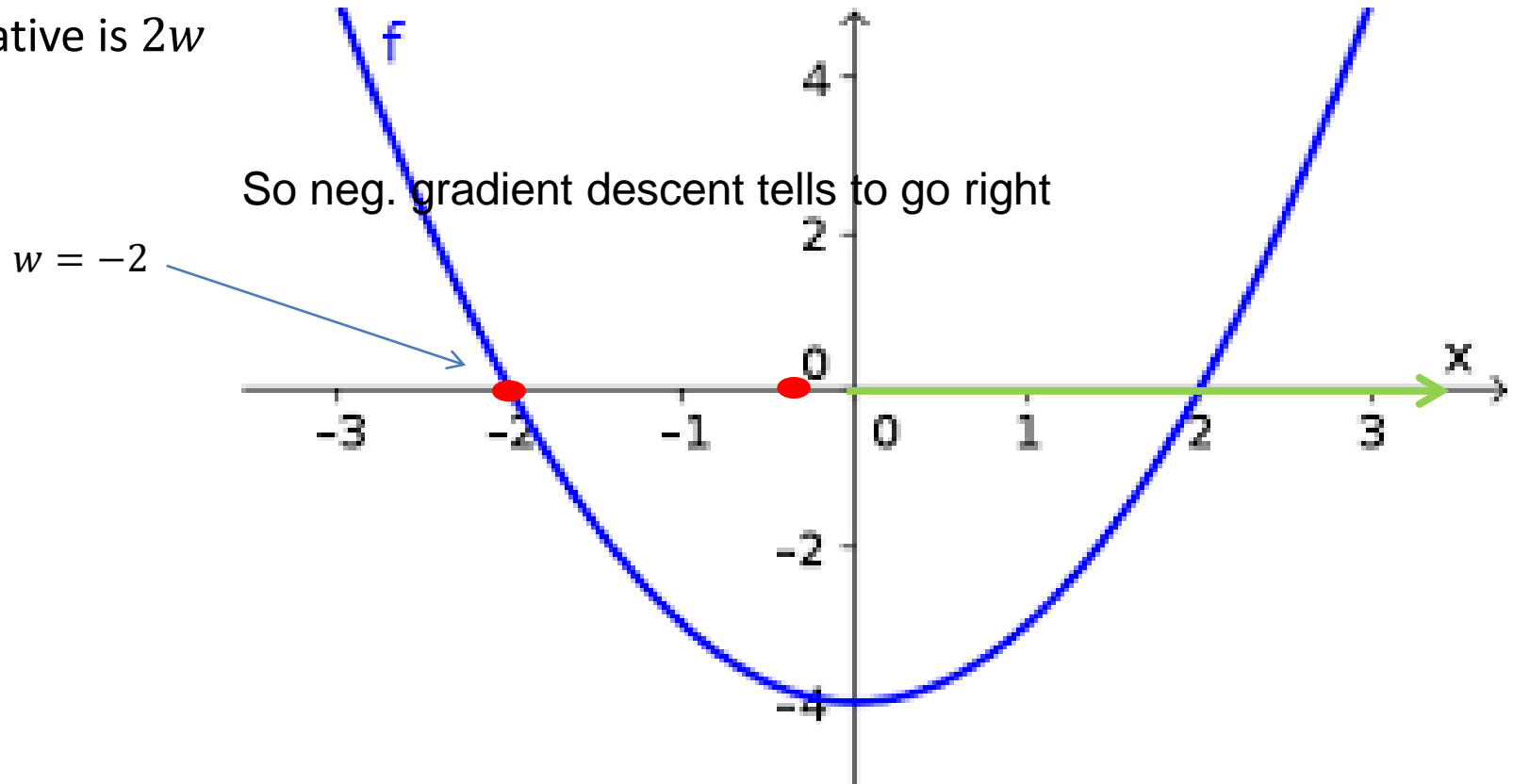
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



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

Gradient descent by example

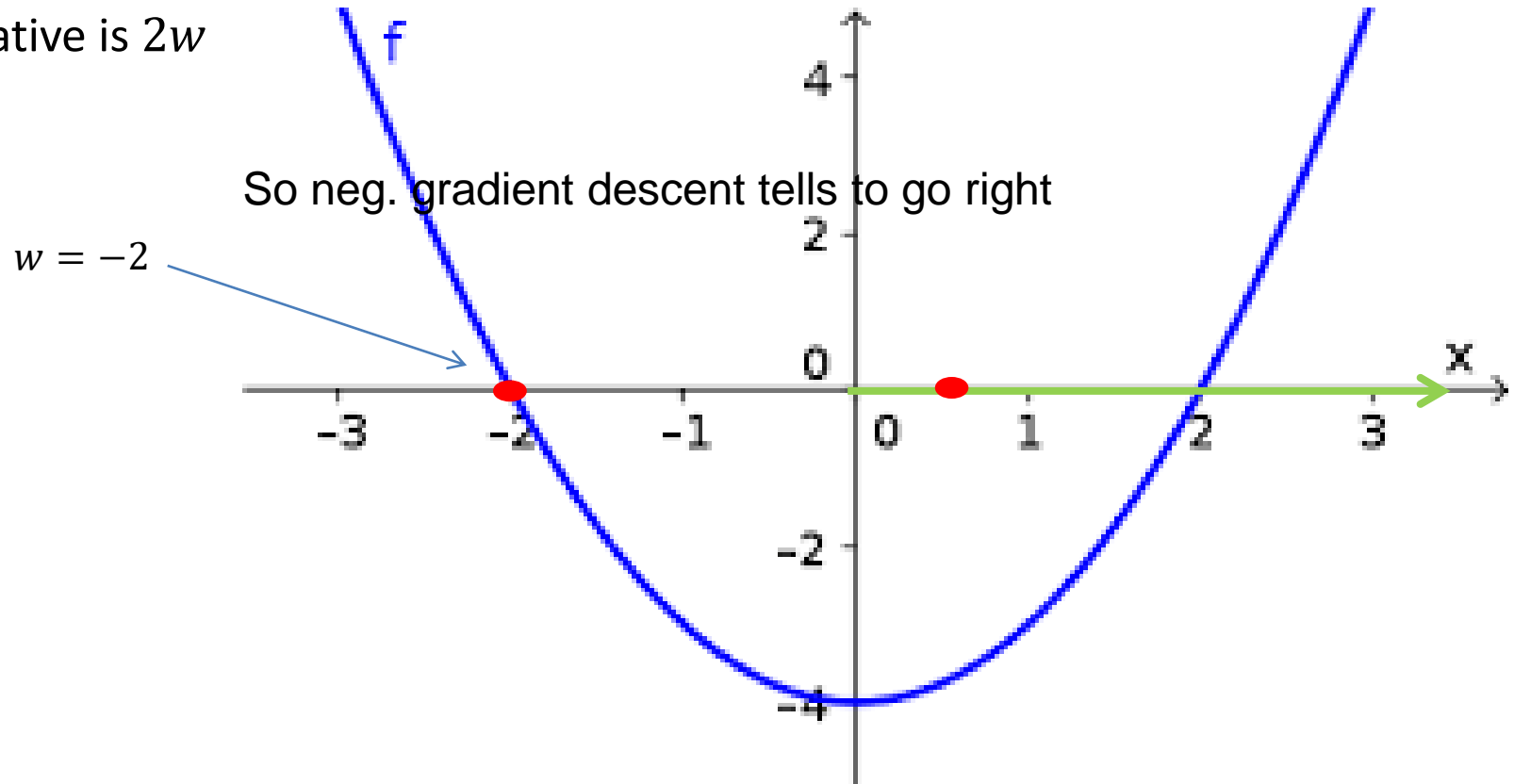
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



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

Gradient descent by example

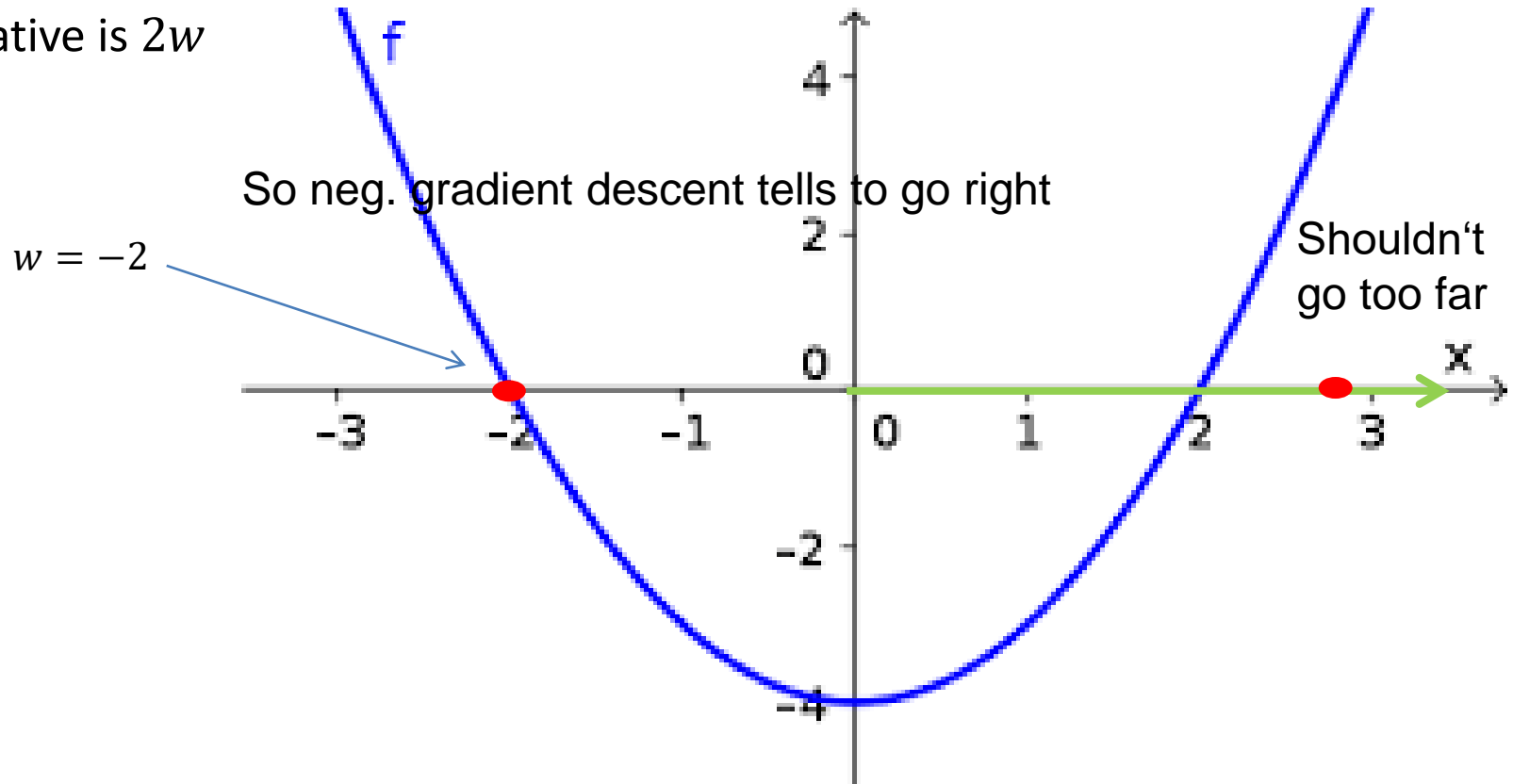
- Consider $F(w) = w^2 - 4$
- Derivative is $2w$



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

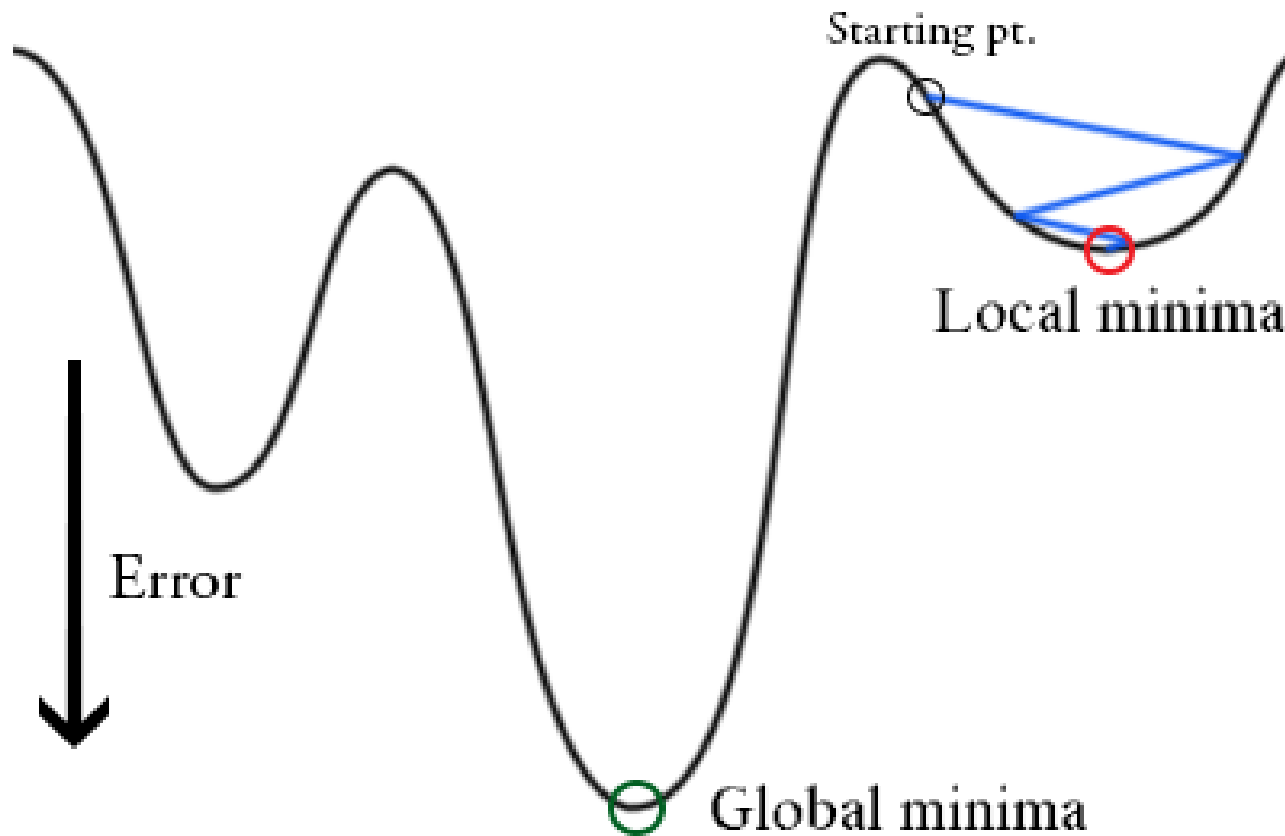
Gradient descent by example

- Consider $F(w) = w^2 - 4$
- 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 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
 - Rarely used for training neural networks

- **Given data:** $(x_1, t_1), \dots, (x_n, t_n)$

- **Error- / Loss-Function:** $\ell(y, t)$

- **Objective:**

$$\min_{\mathbf{w}} F(\mathbf{w}) = \min_{\mathbf{w}} \sum_i \ell(f(x_i; \mathbf{w}), t_i)$$

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

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

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

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

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \sum_i \nabla \ell(f(x_i; \mathbf{w}), t_i)$$

- **Given data:** $(x_1, t_1), \dots, (x_n, t_n)$

- **Error- / Loss-Function:** $\ell(y, t)$

- **Objective:**

$$\min_{\mathbf{w}} F(\mathbf{w}) = \min_{\mathbf{w}} \sum_i \ell(f(x_i; \mathbf{w}), t_i)$$

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

- **Online learning:**
 - **Approximate** ∇F by computing it at only one data point

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla \ell(f(x_i; \mathbf{w}), t_i) \quad \text{for all } i = 1, \dots, n$$

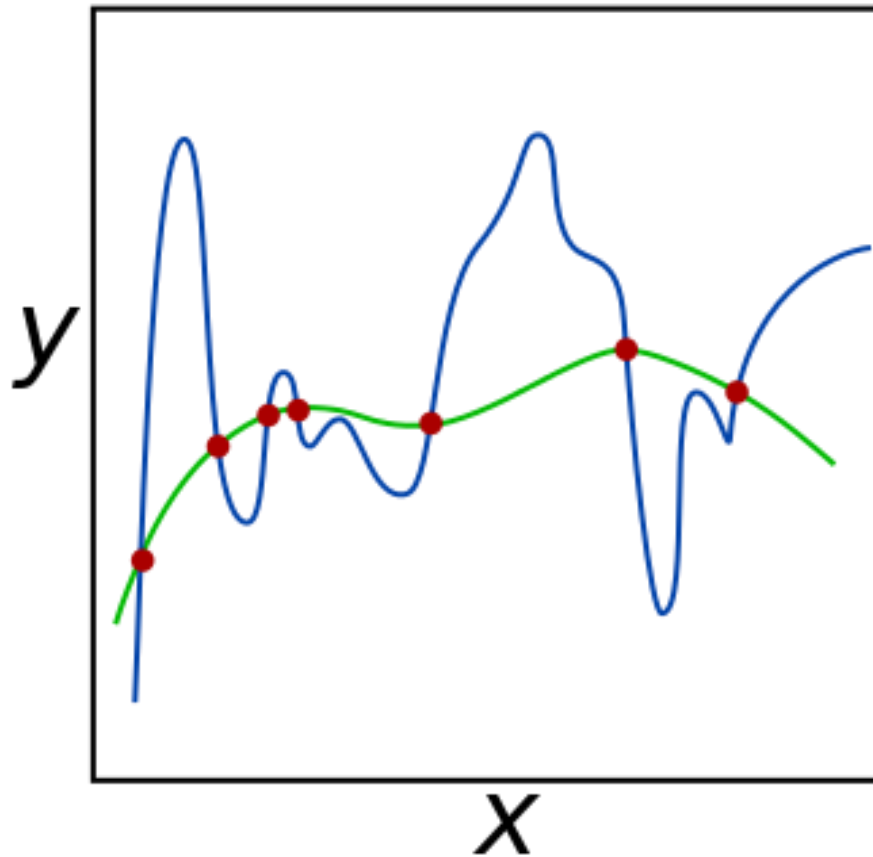
- Intermediate solution between batch and online learning
- Select $k < n$ datapoints (randomly) for gradient computation
- $k = 1 \rightarrow$ online learning, $k = n \rightarrow$ batch learning
- *Advantage:*
 - Computing loss gradient on all datapoints computationally expensive
 - Mini-batch learning converges faster to a good solution than batch learning
- *Disadvantage:*
 - Unclear how to choose k
 - Smaller k 's may lead to better solutions (generalize better)
- Mini-batch learning is also known as (a.k.a) **stochastic gradient descent**
 - **SGD**

Beyond gradient descent

- Adapt learning rate α over time: $\alpha \rightarrow \alpha_t$
- Include *momentum*
 - smooth with past gradients
- Include a *regularization term*

- Minimize
 - $\min_{\{\mathbf{w} \in \mathbb{R}^n\}} F(\mathbf{w}) + \gamma R(\mathbf{w})$
 - for $\gamma \geq 0$
- Choose, e.g.,
 - $R(\mathbf{w}) = \|\mathbf{w}\|^2 = \sum_i w_i^2$ (L2 regularization)
 - $R(\mathbf{w}) = |\mathbf{w}| = \sum_i |w_i|$ (L1 regularization)
- Motivation: Occam's razor
 - choose simpler solutions over more complicated ones

Modifications: Regularization



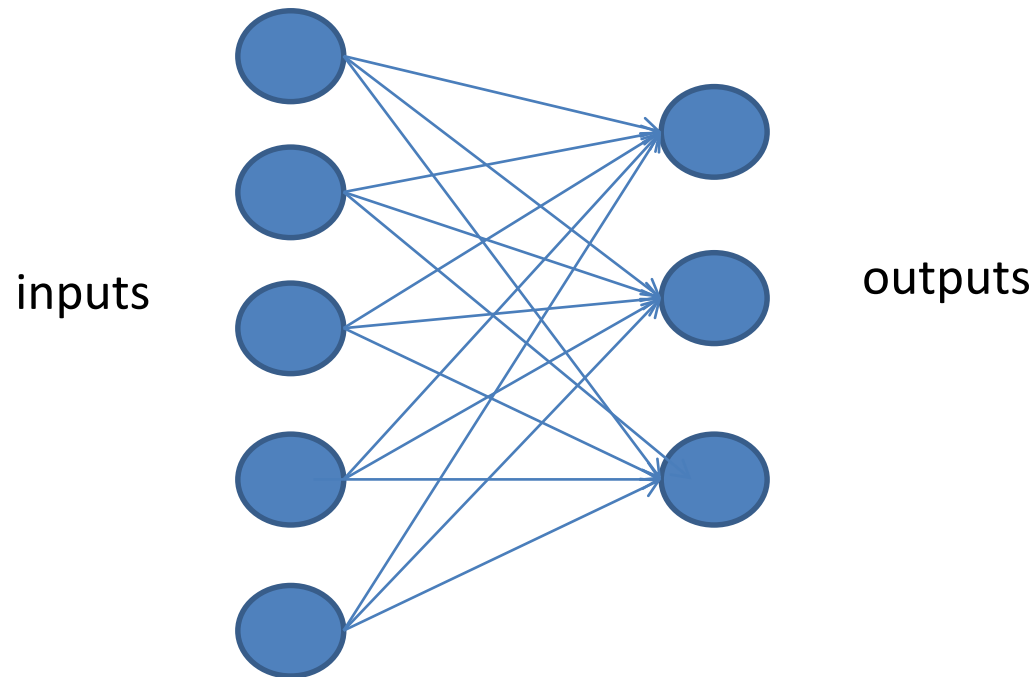
- From [https://en.wikipedia.org/wiki/Regularization_\(mathematics\)](https://en.wikipedia.org/wiki/Regularization_(mathematics))

- More advanced optimizers, beyond SGD, 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 remedy: First train with Adam, fine-tune with SGD
- Great overview: <http://runder.io/optimizing-gradient-descent/index.htm>

Deep Networks – Terminology (Recap)

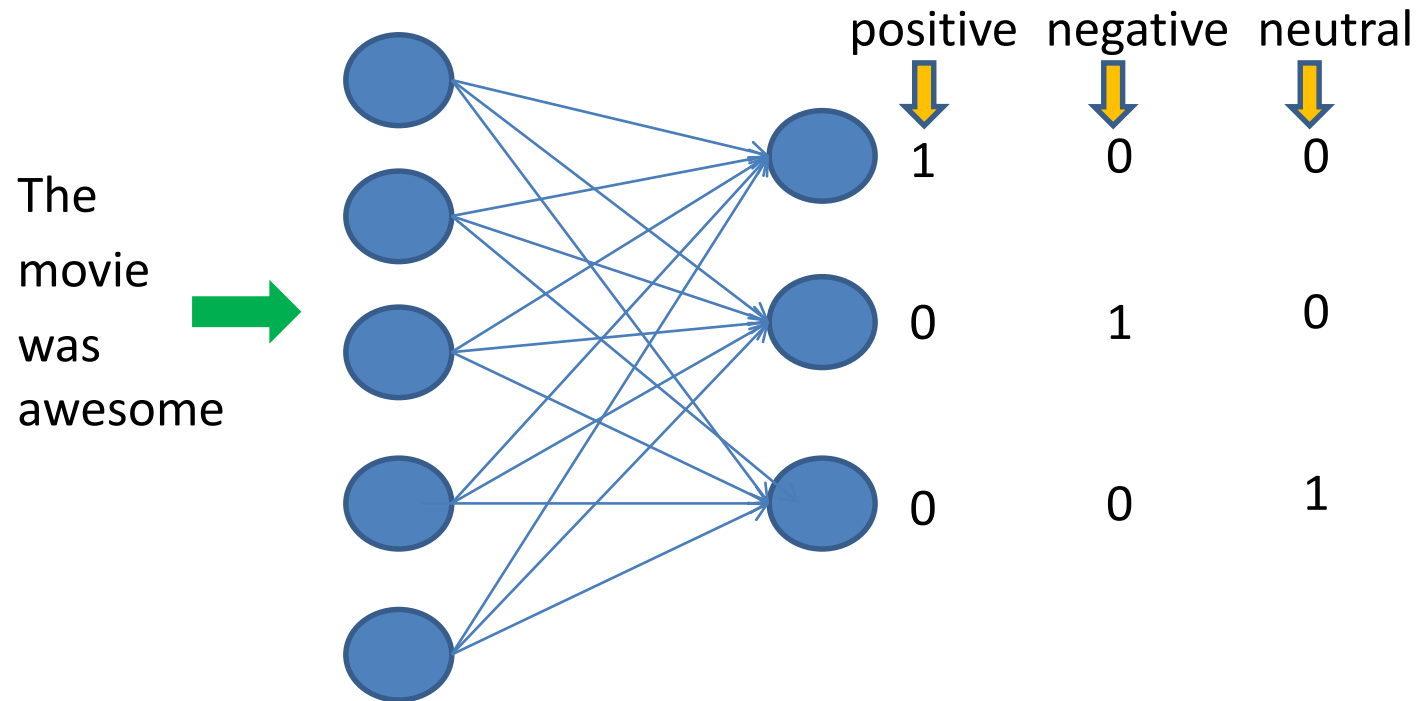
More complex neural networks

- Several output neurons instead of a single output neuron



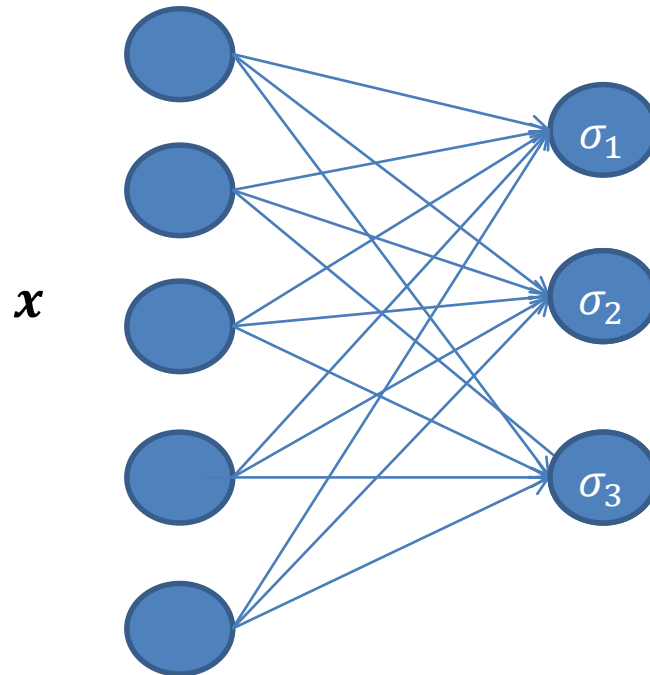
More complex neural networks

- Several output neurons instead of a single output neuron



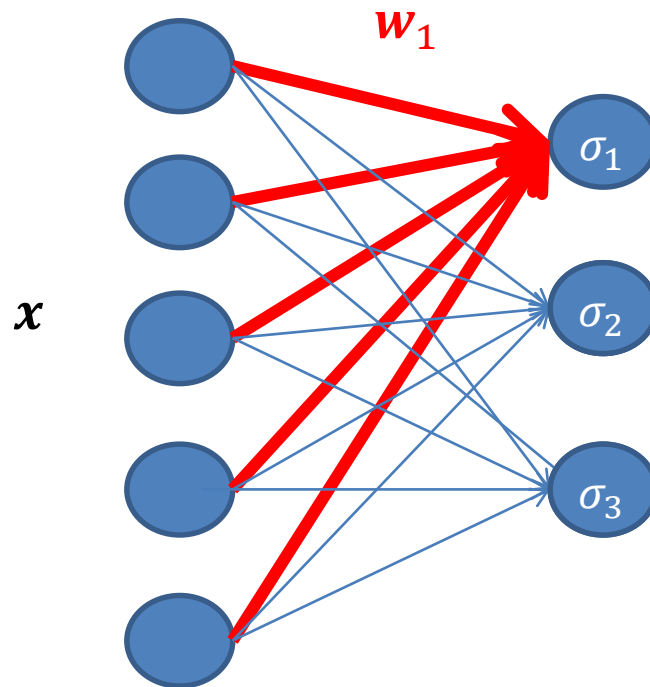
More complex neural networks

- Several output neurons instead of a single output neuron



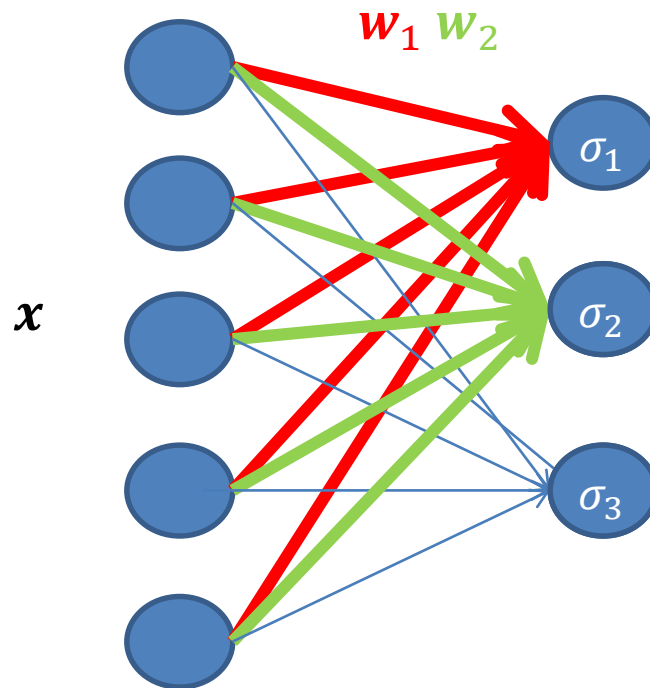
More complex neural networks

- Several output neurons instead of a single output neuron



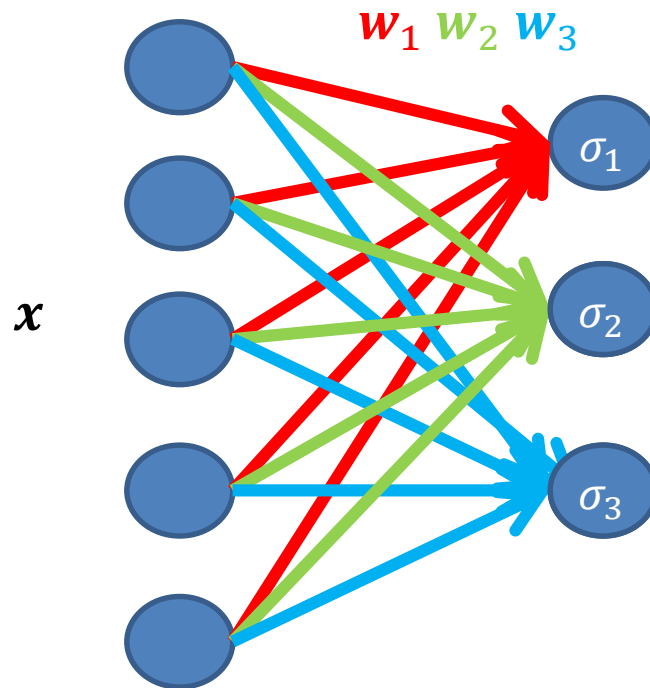
More complex neural networks

- Several output neurons instead of a single output neuron



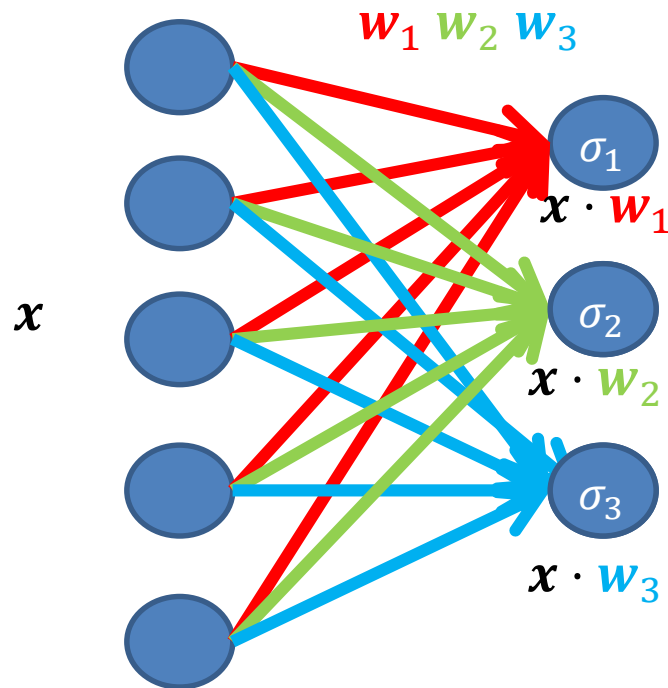
More complex neural networks

- Several output neurons instead of a single output neuron



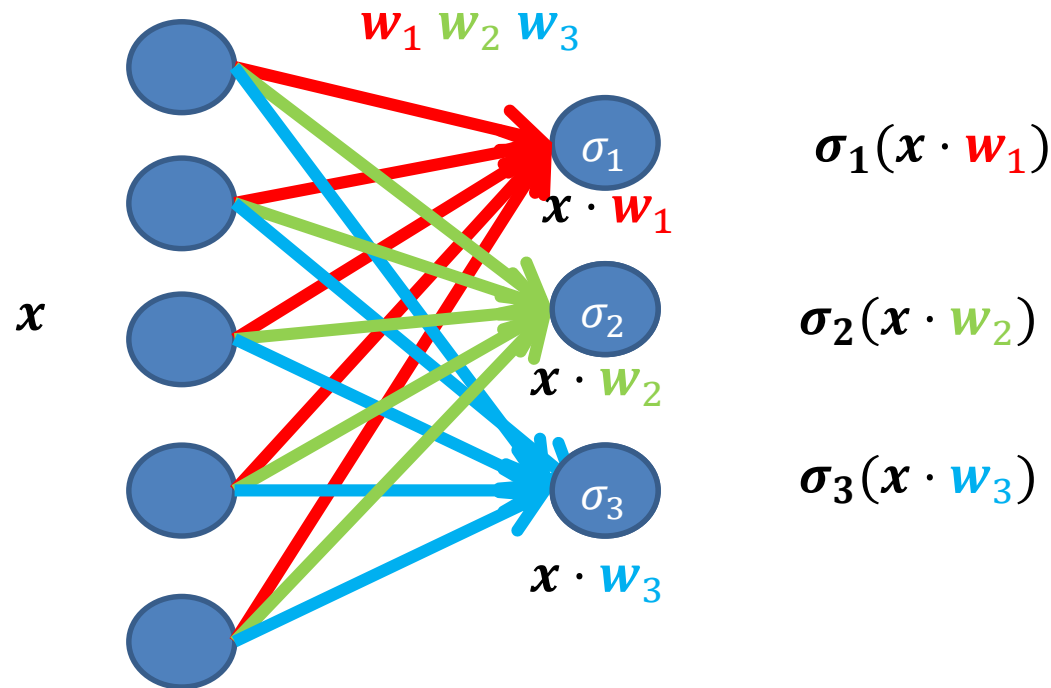
More complex neural networks

- Several output neurons instead of a single output neuron



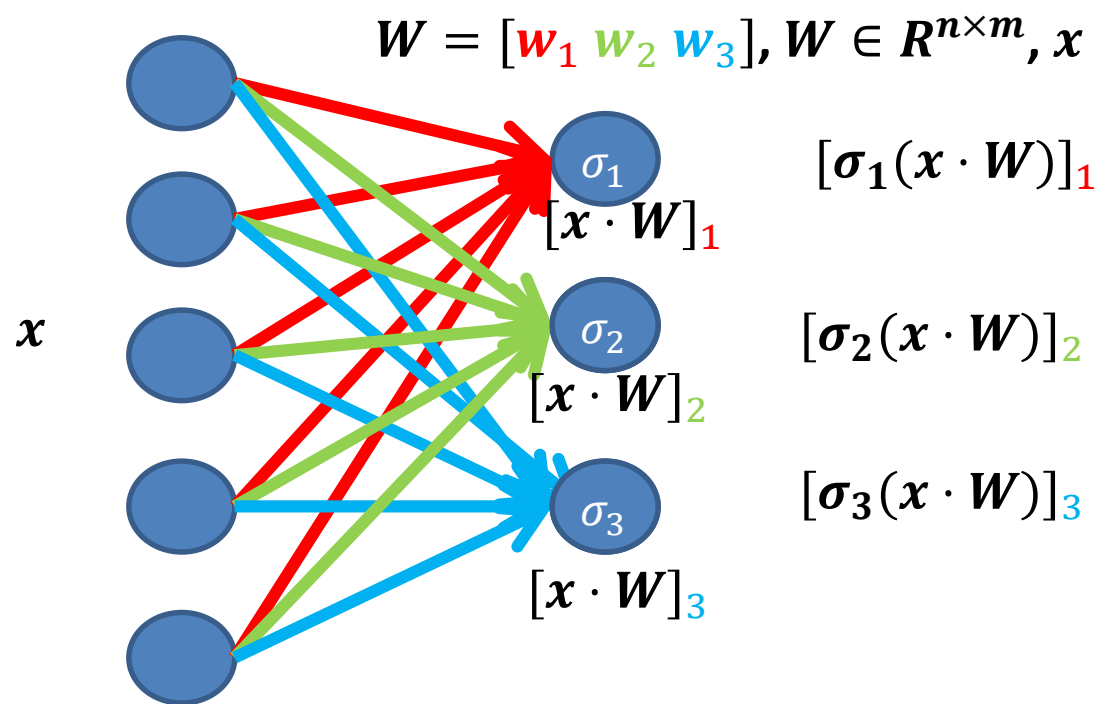
More complex neural networks

- Several output neurons instead of a single output neuron



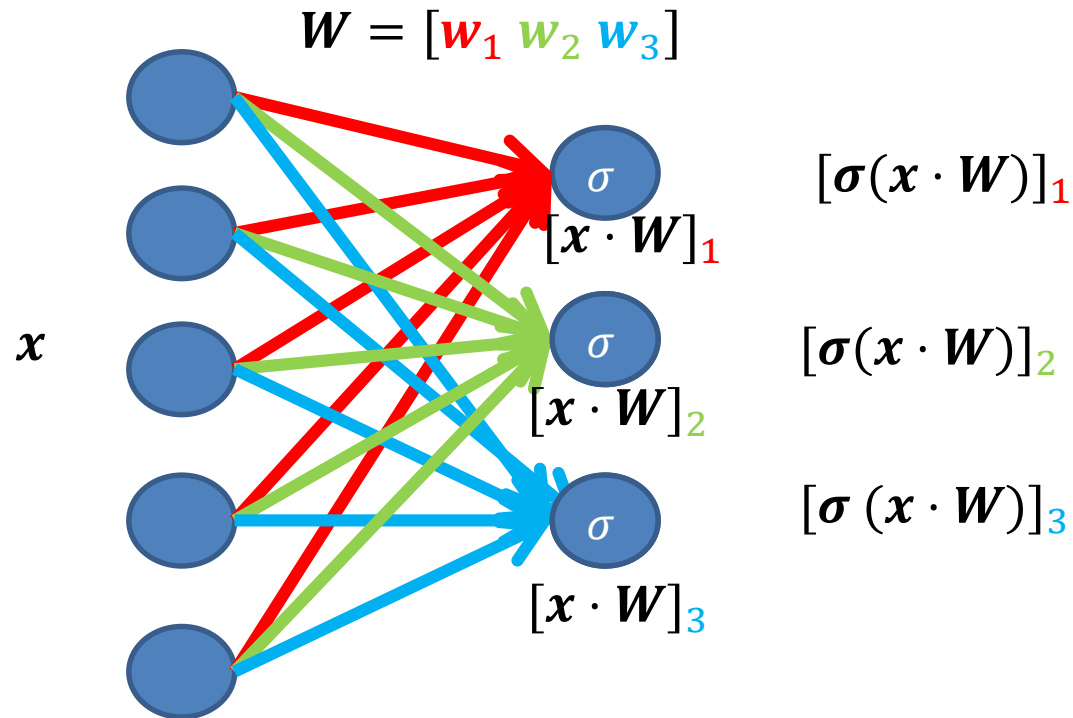
More complex neural networks

- Several output neurons instead of a single output neuron



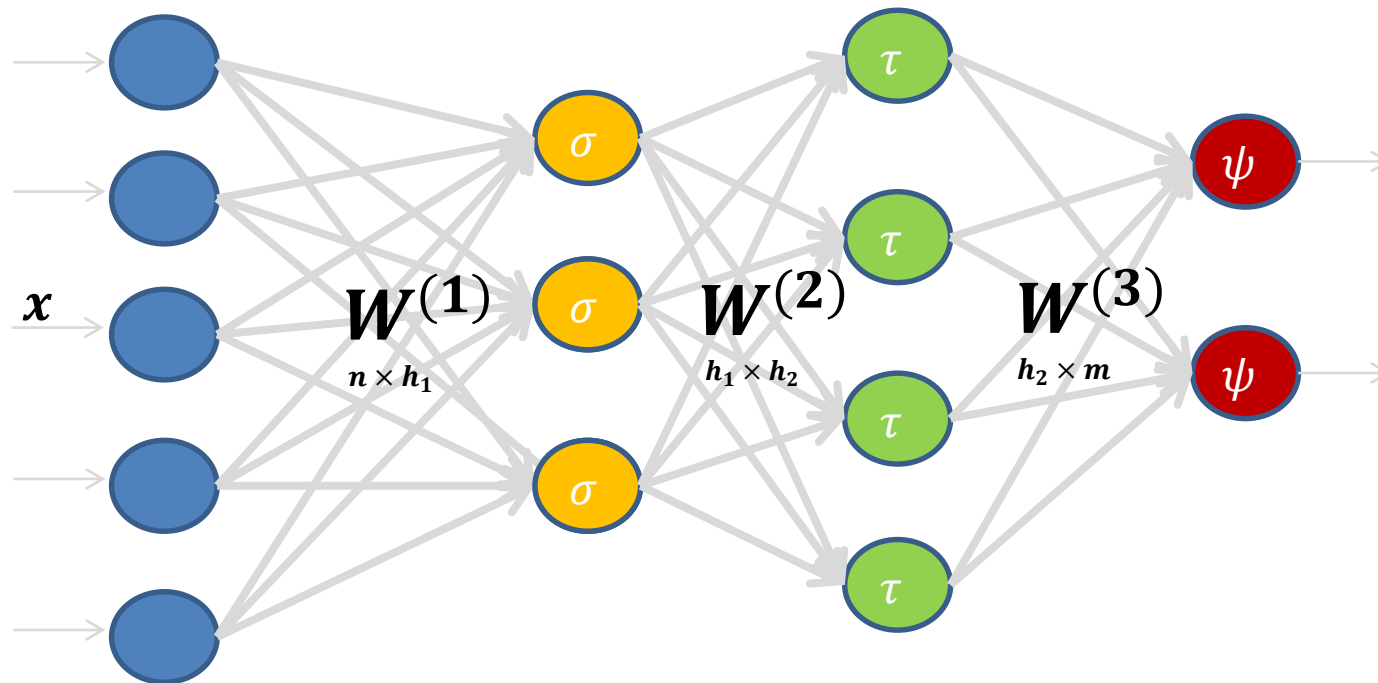
More complex neural networks

- Several output neurons instead of a single output neuron



More complex neural networks

- 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

Feed-forward NN
a.k.a.
MLP

- $f: \mathbf{R}^n \rightarrow \mathbf{R}^m,$
- with parameter (matrices) $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \dots, \mathbf{W}^{(H)}$
- and non-linearities $\sigma_1, \sigma_2, \dots, \sigma_H$
- where

- $\mathbf{z}^{(1)} = \mathbf{x} \cdot \mathbf{W}^{(1)}$
- $\mathbf{y}^{(1)} = \sigma_1(\mathbf{z}^{(1)})$
- $\mathbf{z}^{(2)} = \mathbf{y}^{(1)} \cdot \mathbf{W}^{(2)}$
-
- $\mathbf{z}^{(H)} = \mathbf{y}^{(H-1)} \cdot \mathbf{W}^{(H)}$
- $\mathbf{y} = \mathbf{y}^{(H)} = \sigma_H(\mathbf{z}^{(H)})$

More complex neural networks

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

Why do we need hidden layers?

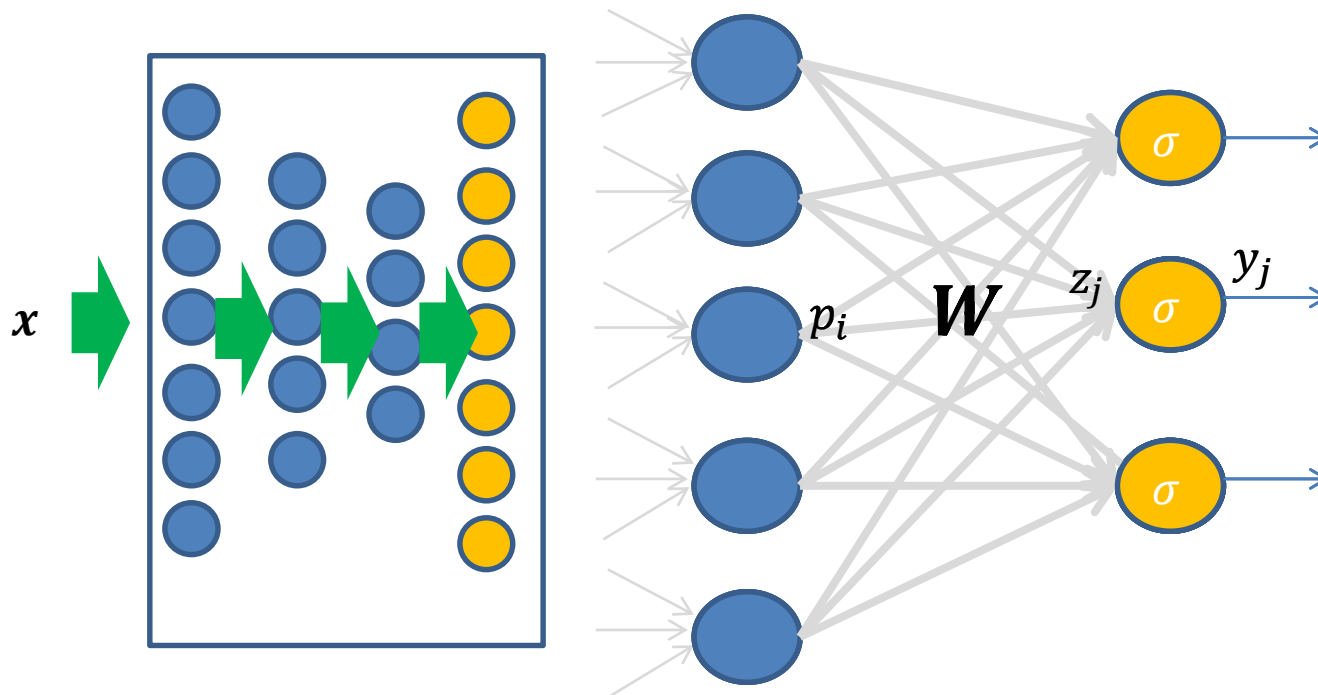
- Hidden layers can learn useful intermediate representations of the data
 - Helps learning
 - A good organization of hidden layers can make learning much faster
- Perceptron cannot even learn the XOR function
 - In contrast, MLP with one hidden layer is a *universal approximator*
 - See Cybenko 1989, 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

Backpropagation

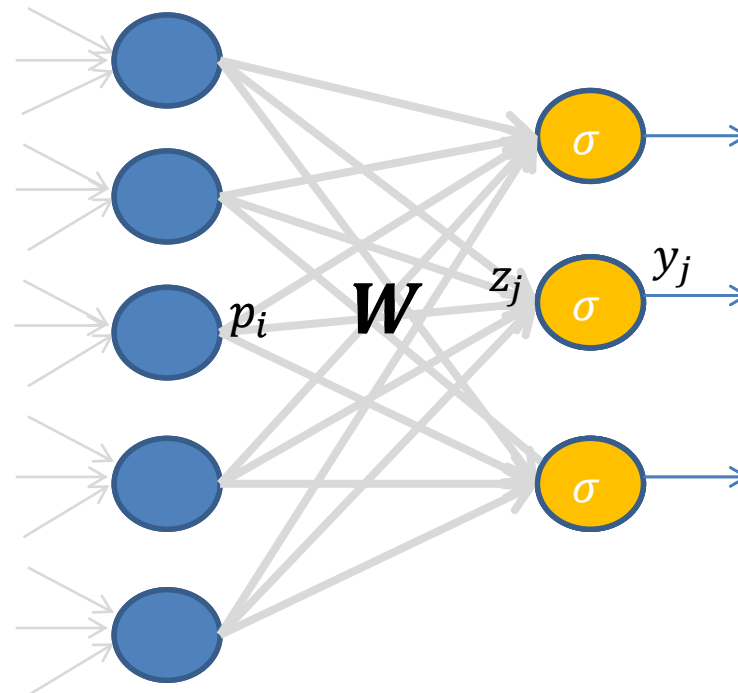
- When we want to use gradient descent for neural network learning, we need gradients over our loss
- For perceptrons, we could easily derive $\nabla \ell$ ourselves (by hand)
- For general MLP, we cannot derive $\nabla \ell$ so easily:
 - How to derive $\nabla \ell$ in these situations is the scope of the **backprop algorithm**

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

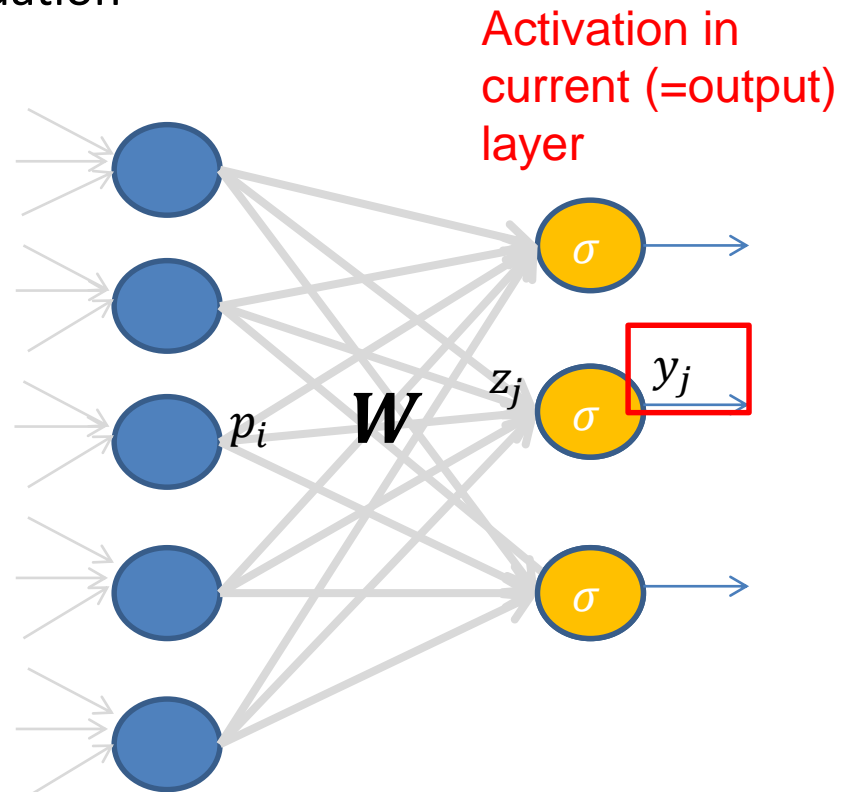
- Consider the following model situation



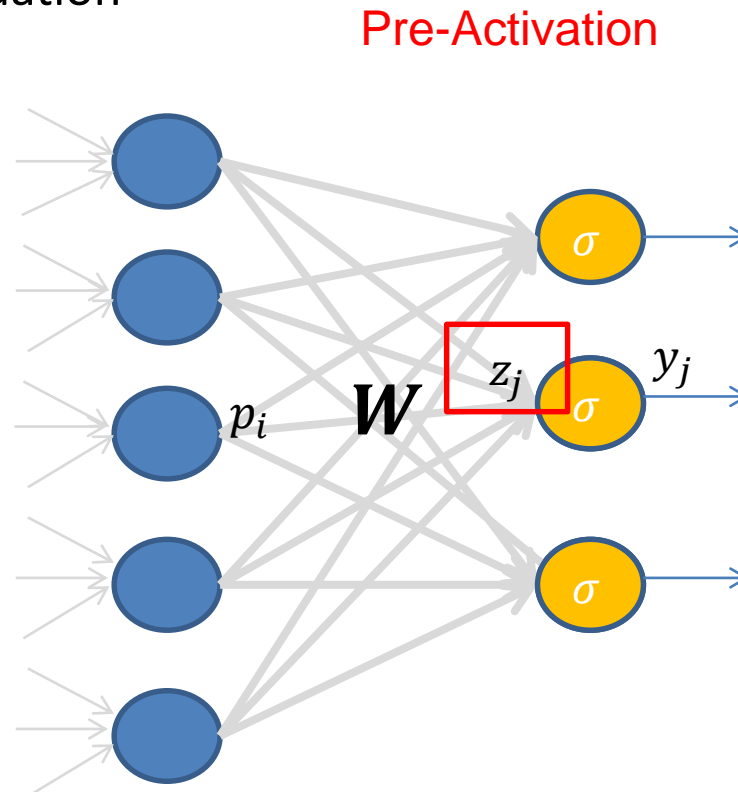
- Consider the following model situation



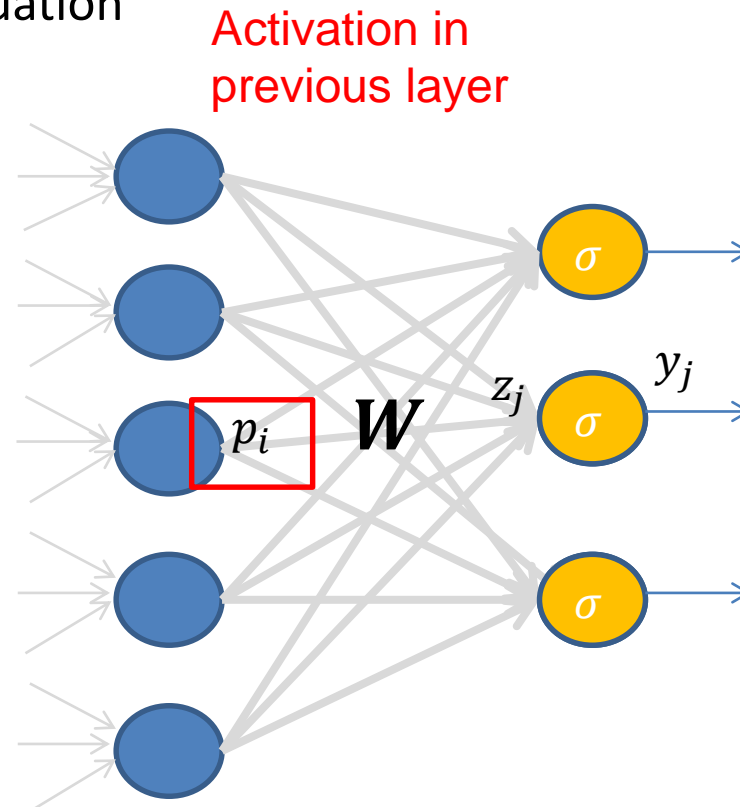
- Consider the following model situation



- 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 (\mathbf{x}, \mathbf{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
 - $L(\boldsymbol{\theta}) = \sum_{(x,t)} \ell(\mathbf{y}, \mathbf{t})$
 - The output $\mathbf{y} \in R^m$ is determined by some (deep) MLP
- We focus on minimizing
 - $L(\boldsymbol{\theta}) = \ell(\mathbf{y}, \mathbf{t})$
 - for notational convenience
- We focus on (multi-dim) square loss, but you can substitute other loss functions and derive analogous results
 - $\ell(\mathbf{y}, \mathbf{t}) = \|\mathbf{y} - \mathbf{t}\|^2$

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

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

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

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
 - $\delta := \partial \ell / \partial q_i$
 - q_i is activation of some neuron
 - “how much does my loss change when activation in some neuron (in some layer) changes”
 - Then we relate δ to $\frac{\partial \ell}{\partial w_{ij}}$

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

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

- We start by asking ourselves what

- $\frac{\partial \ell}{\partial p_i}$ is

- We find

- $\frac{\partial \ell}{\partial p_i} = \sum_j \frac{\partial e_j}{\partial p_i}$

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

- We start by asking ourselves what

- $\frac{\partial \ell}{\partial p_i}$ is

- We find

- $\frac{\partial \ell}{\partial p_i} = \sum_j \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial p_i}$

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

- We start by asking ourselves what

- $\frac{\partial \ell}{\partial p_i}$ is

Chain rule

- We find

- $\frac{\partial \ell}{\partial p_i} = \sum_j \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial p_i}$

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

- We start by asking ourselves what

- $\frac{\partial \ell}{\partial p_i}$ is

- We find

- $\frac{\partial \ell}{\partial p_i} = \sum_j \frac{\partial e_j}{\partial y_j} \frac{\partial y_j}{\partial p_i}$ (note that $\frac{\partial e_j}{\partial y_j} = \frac{\partial \ell}{\partial y_j}$)

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

$$\begin{aligned} \ell &= e_1(y_1) + e_2(y_2) \\ &+ \dots + e_j(y_j) + \dots \end{aligned}$$

- We start by asking ourselves what

- $\frac{\partial \ell}{\partial p_i}$ is

- We find

- $\frac{\partial \ell}{\partial p_i} = \sum_j \frac{\partial \ell}{\partial y_j} \frac{\partial y_j}{\partial p_i}$

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

- We start by asking ourselves what

- $\frac{\partial \ell}{\partial p_i}$ is

- We find

- $\frac{\partial \ell}{\partial p_i} = \sum_j \frac{\partial \ell}{\partial y_j} \frac{\partial y_j}{\partial p_i}$

- 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
 - $\frac{\partial \ell}{\partial p_i}$ is
- We find
 - $\frac{\partial \ell}{\partial p_i} = \sum_j \frac{\partial \ell}{\partial y_j} \frac{\partial y_j}{\partial p_i}$
- 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:
 - $z_j = \mathbf{p} \cdot \mathbf{w}_j$
 - $\frac{\partial z_j}{\partial p_i} = [\mathbf{w}_j]_i$ (note that $[\mathbf{w}_j]_i = w_{ij}$)

- We start by asking ourselves what

- $\frac{\partial \ell}{\partial p_i}$ is

- We find

- $\frac{\partial \ell}{\partial p_i} = \sum_j \frac{\partial \ell}{\partial y_j} \frac{\partial y_j}{\partial p_i}$

- 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:

- $z_j = \mathbf{p} \cdot \mathbf{w}_j$

- $\frac{\partial z_j}{\partial p_i} = w_{ij}$

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

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

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



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

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



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

- That's a great result

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



We expressed loss 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

- $\frac{\partial \ell}{\partial y_j} = -2(t_j - y_j) = 2(y_j - t_j)$

$$\ell(\mathbf{y}, \mathbf{t}) = \|\mathbf{y} - \mathbf{t}\|^2 = \sum_j (t_j - y_j)^2$$

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



We expressed loss 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
 - $\frac{\partial \ell}{\partial y_j} = 2(y_j - t_j)$
- But from this we know the values at the layer (lastLayer-1) by our formula above

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



We expressed loss 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
 - $\frac{\partial \ell}{\partial y_j} = 2(y_j - t_j)$
- 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
-

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



We expressed loss 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
 - $\frac{\partial \ell}{\partial y_j} = 2(y_j - t_j)$
- 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
-

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



We expressed loss 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
 - $\frac{\partial \ell}{\partial y_j} = 2(y_j - t_j)$
- 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!

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



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

- But we're looking for

- $\frac{\partial \ell}{\partial u_{ik}}$ for all of the weight matrices $\mathbf{U} = \mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(3)}, \dots$

But we're looking for derivatives wrt. weights!

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



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

- But we're looking for

- $\frac{\partial \ell}{\partial u_{ik}}$ for all of the weight matrices $\mathbf{U} = \mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(3)}, \dots$

- Fortunately, we have the relation

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

- Here, m_k is the output/activation at the layer corresponding to matrix \mathbf{U}
- l is the input for that layer
- z_k is the pre-activation of m_k , i.e., $m_k = \sigma(z_k)$

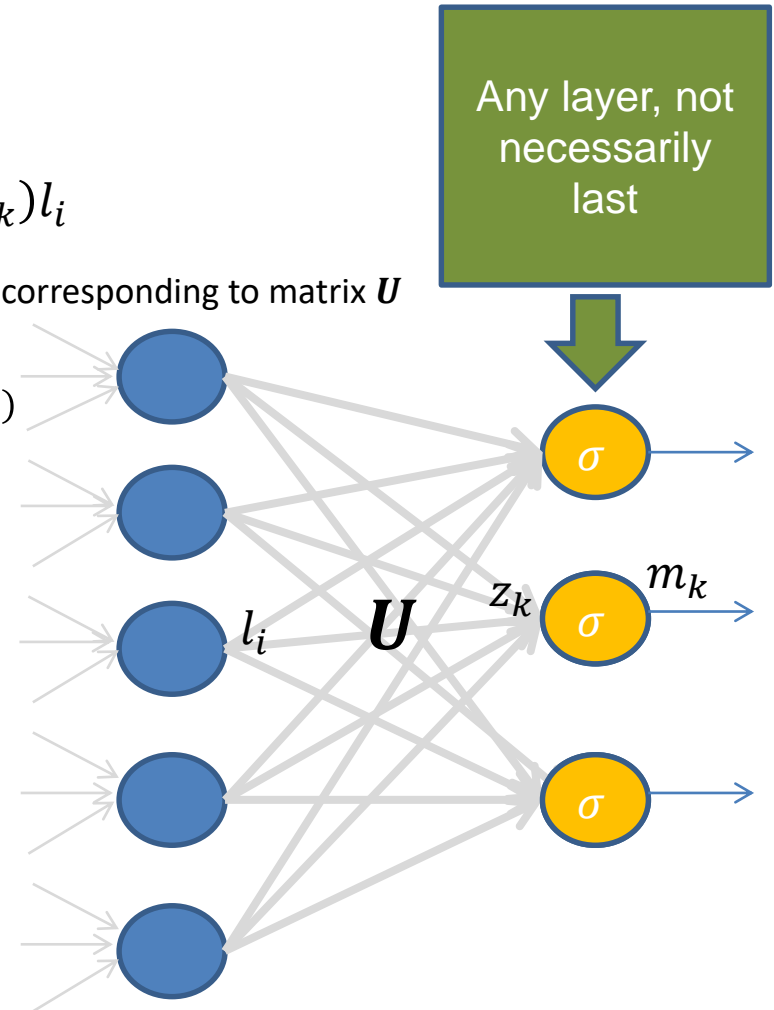


But we're looking for derivatives wrt. weights!

- Fortunately, we have the relation

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

- Here, m_k is the output/activation at the layer corresponding to matrix U
 - l is the input for that layer
 - z_k is the pre-activation of m_k , i.e., $m_k = \sigma(z_k)$



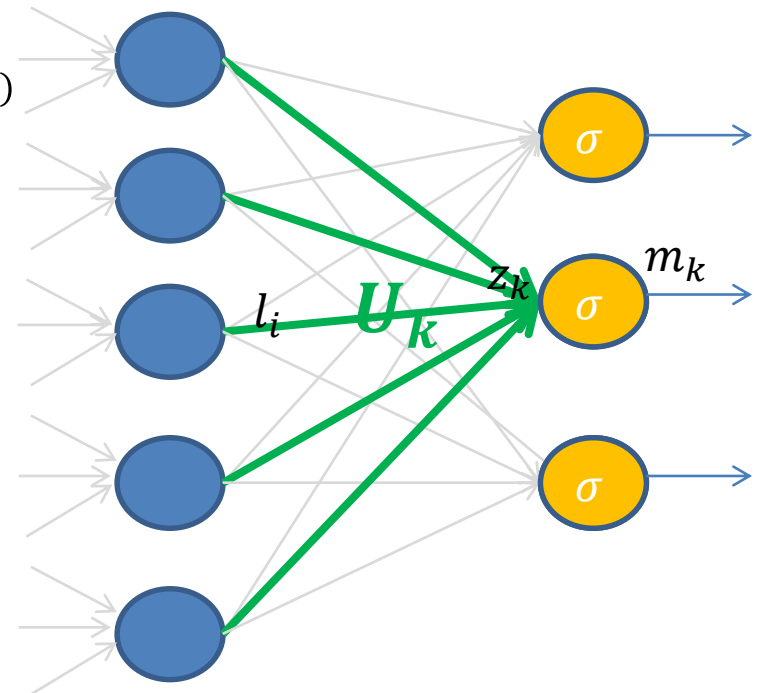
But we're looking for derivatives wrt. weights!

- Fortunately, we have the relation

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

- Here, m_k is the output/activation at the layer corresponding to matrix U
- l is the input for that layer
- z_k is the pre-activation of m_k , i.e., $m_k = \sigma(z_k)$

$$U = [U_1 \cdots U_k \cdots]$$



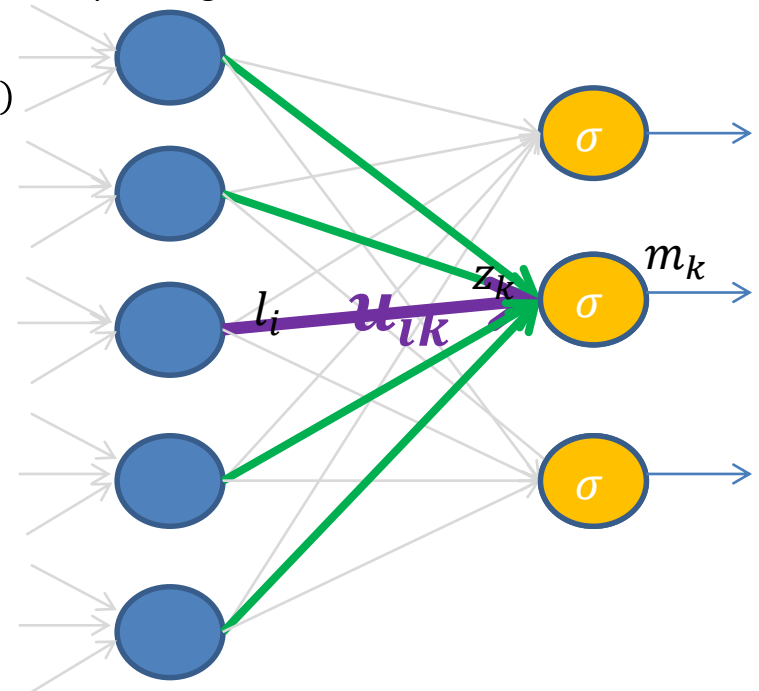
But we're looking for derivatives wrt. weights!

- Fortunately, we have the relation

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

- Here, m_k is the output/activation at the layer corresponding to matrix U
- l is the input for that layer
- z_k is the pre-activation of m_k , i.e., $m_k = \sigma(z_k)$

$$U = [U_1 \cdots U_k \cdots]$$



But we're looking for derivatives wrt. weights!

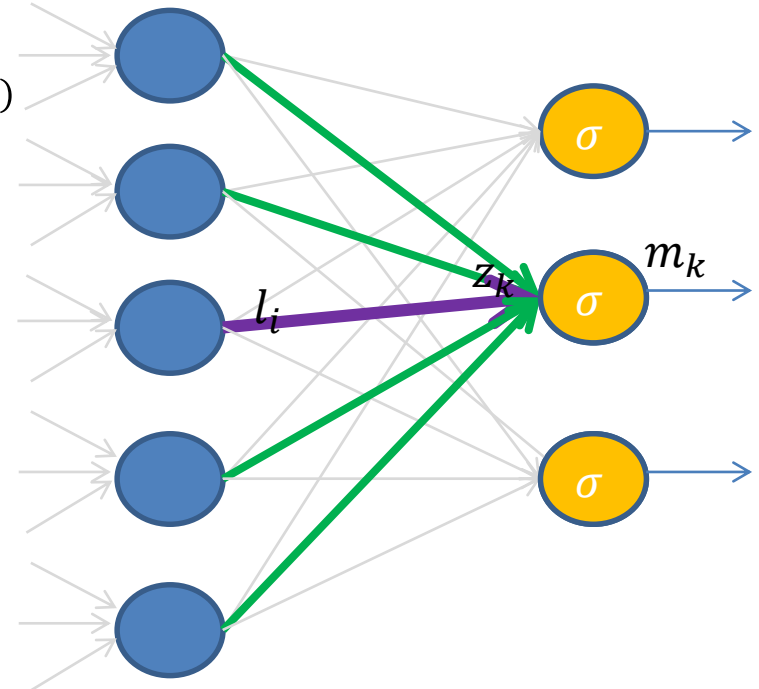
Chain rule

- Fortunately, we have the relation

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

- Here, m_k is the output/activation at the layer corresponding to matrix U
- l is the input for that layer
- z_k is the pre-activation of m_k , i.e., $m_k = \sigma(z_k)$

$$U = [U_1 \cdots U_k \cdots]$$



But we're looking for derivatives wrt. weights!

- Fortunately, we have the relation

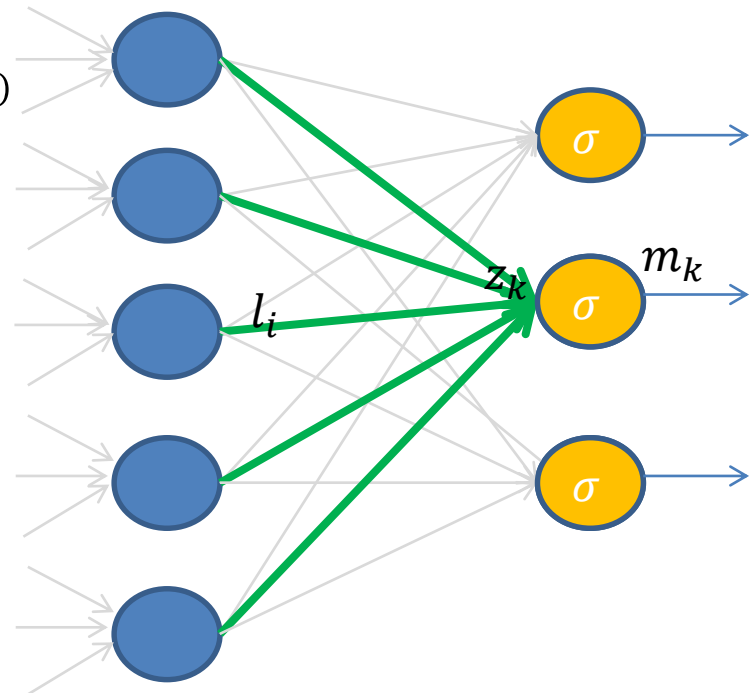
$$\frac{\partial \ell}{\partial u_{ik}} = \frac{\partial \ell}{\partial z_k} \frac{\partial z_k}{\partial u_{ik}} = \frac{\partial \ell}{\partial m_k} \frac{\partial m_k}{\partial z_k} \boxed{l_i} = \frac{\partial \ell}{\partial m_k} \sigma'(z_k) l_i$$

- Here, m_k is the output/activation at the layer corresponding to matrix U
- l is the input for that layer
- z_k is the pre-activation of m_k , i.e., $m_k = \sigma(z_k)$

$$U = [U_1 \cdots \mathbf{U}_k \cdots]$$



$$\begin{aligned} z_k &= l \cdot U_k \\ &= \sum_i l_i (U_k)_i \\ &= \sum_i l_i u_{ik} \end{aligned}$$



But we're looking for derivatives wrt. weights!

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



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

- But we're looking for

- $\frac{\partial \ell}{\partial u_{ik}}$ for all of the weight matrices $\mathbf{U} = \mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(3)}, \dots$

- Fortunately, we have the relation

This we get from backprop

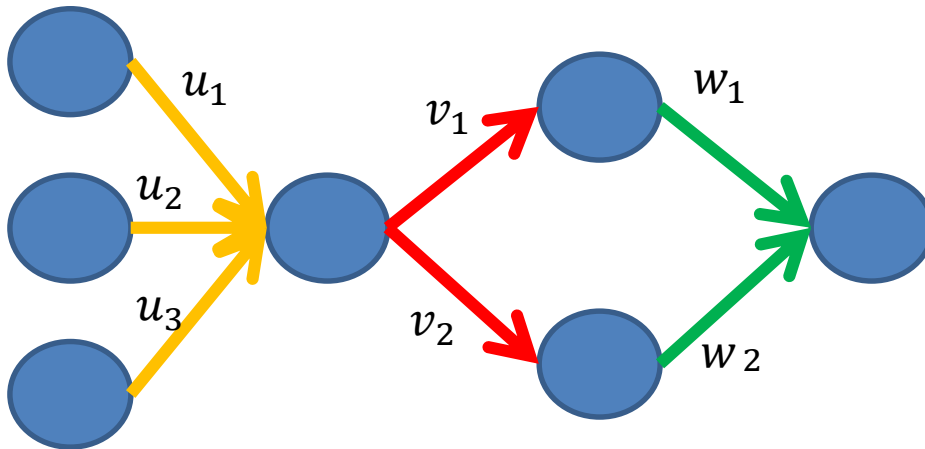
- $\frac{\partial \ell}{\partial u_{ik}} = \frac{\partial \ell}{\partial z_k} \frac{\partial z_k}{\partial u_{ik}} = \frac{\partial \ell}{\partial m_k} \frac{\partial m_k}{\partial z_k} l_i = \boxed{\frac{\partial \ell}{\partial m_k}} \sigma'(z_k) l_i$

- Here, m_k is the output/activation at the layer corresponding to matrix \mathbf{U}
- \mathbf{l} is the input for that layer
- z_k is the pre-activation of m_k , i.e., $m_k = \sigma(z_k)$

- Backpropagation is a recursive algorithm for determining loss derivatives
- Starts at the outer layer
- Propagates loss derivative signal 'backwards'
 - By expressing derivatives in one layer in terms of the next layer derivatives **using the chain rule**
- General technique for calculating derivatives of composite functions
 - Modifications of the presented algorithm apply to other mathematical functions, too (not only neural nets!)

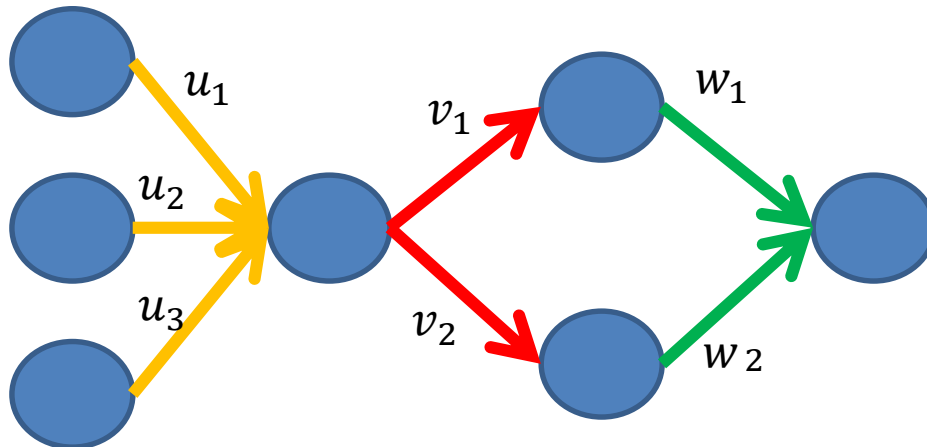
Example

- 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 $\mathbf{u} = (0.2, 0.5, -1)$, $\mathbf{v} = (0.9, -0.5)$, $\mathbf{w} = (0.2, -5)$

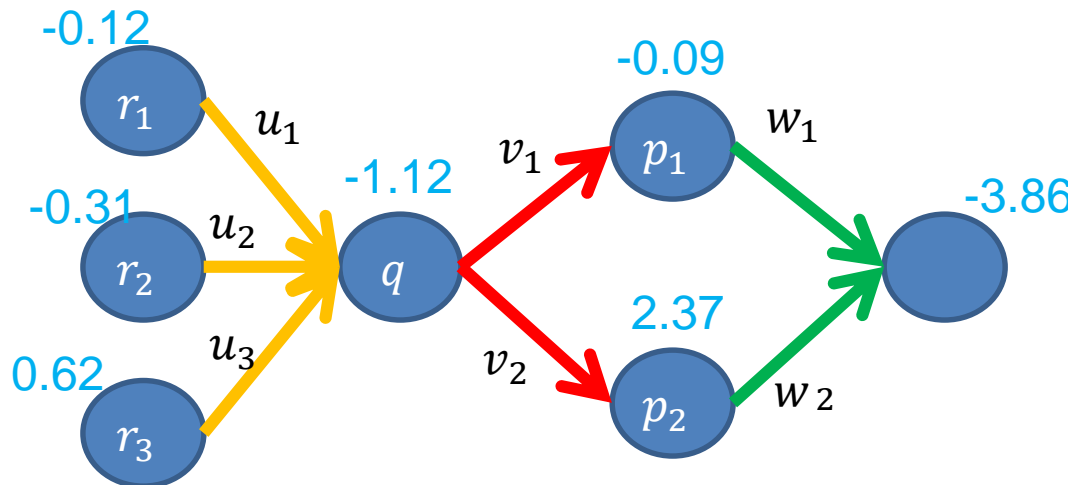


- Assume that $\mathbf{x} = (1, 0, 1)$ and $t = 1$
- Square loss: $(t - y)^2$

- 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 $\mathbf{u} = (0.2, 0.5, -1)$, $\mathbf{v} = (0.9, -0.5)$, $\mathbf{w} = (0.2, -5)$



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

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

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

Example – gradient check

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

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

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

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

- and at $\mathbf{u}, \mathbf{v}, \mathbf{w}$
- Then, we compute

$$\frac{1}{h} \cdot (\ell(\mathbf{x}, t; \mathbf{u}', \mathbf{v}, \mathbf{w}) - \ell(\mathbf{x}, t; \mathbf{u}, \mathbf{v}, \mathbf{w}))$$

(Neural) Language Models

- Language model:

- Assigns a sequence of tokens (words, characters, ...) a probability
- Which denotes likelihood of observing this sequence in text

- Intuitively

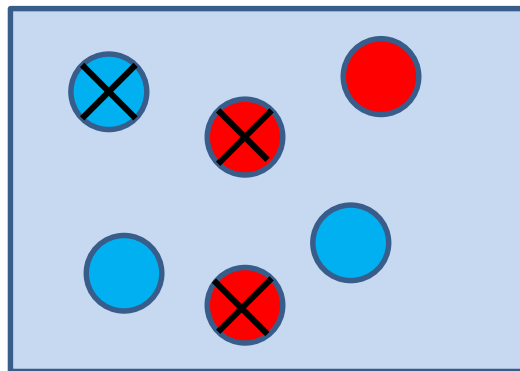
- $\text{Probability}(\text{the cat sat on the mat}) > \text{Probability}(\text{mat sat cat on the the})$

- 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 probab.: $P(A|B) = P(A, B)/P(B)$
 - how likely is it to observe A given B?
- Marginal probability: $P(A)$
 - how likely is it to observe event A?



female

male

X = committed crime

- What is $P(\text{female})$? $1/2$
- What is $P(\text{female, crime})$? $2/6$
- What is $P(\text{female}|\text{crime})$? $2/3$
- What is $P(\text{crime}|\text{female})$? $2/3$
- What is $P(\text{crime}|\text{male})$? $1/3$

- Previously, a common approach was to use **n-gram** language models

- Approximate the true probability P of a stream of tokens

$$\begin{aligned} &P(w_1 w_2 w_3 w_4 \dots) \\ &= P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1w_2) \cdot P(w_4|w_1w_2w_3) \\ &\cdot P(w_5|w_1w_2w_3w_4) \dots \end{aligned}$$

- by an n-gram model, where:

$$P(w_t|w_1 \dots w_{t-1}) \approx P(w_t|w_{t-n+1} \dots w_{t-1})$$

Complete history

History of size n-1

- For example, 1-gram model (unigram)

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

- 2-gram model (bigram)

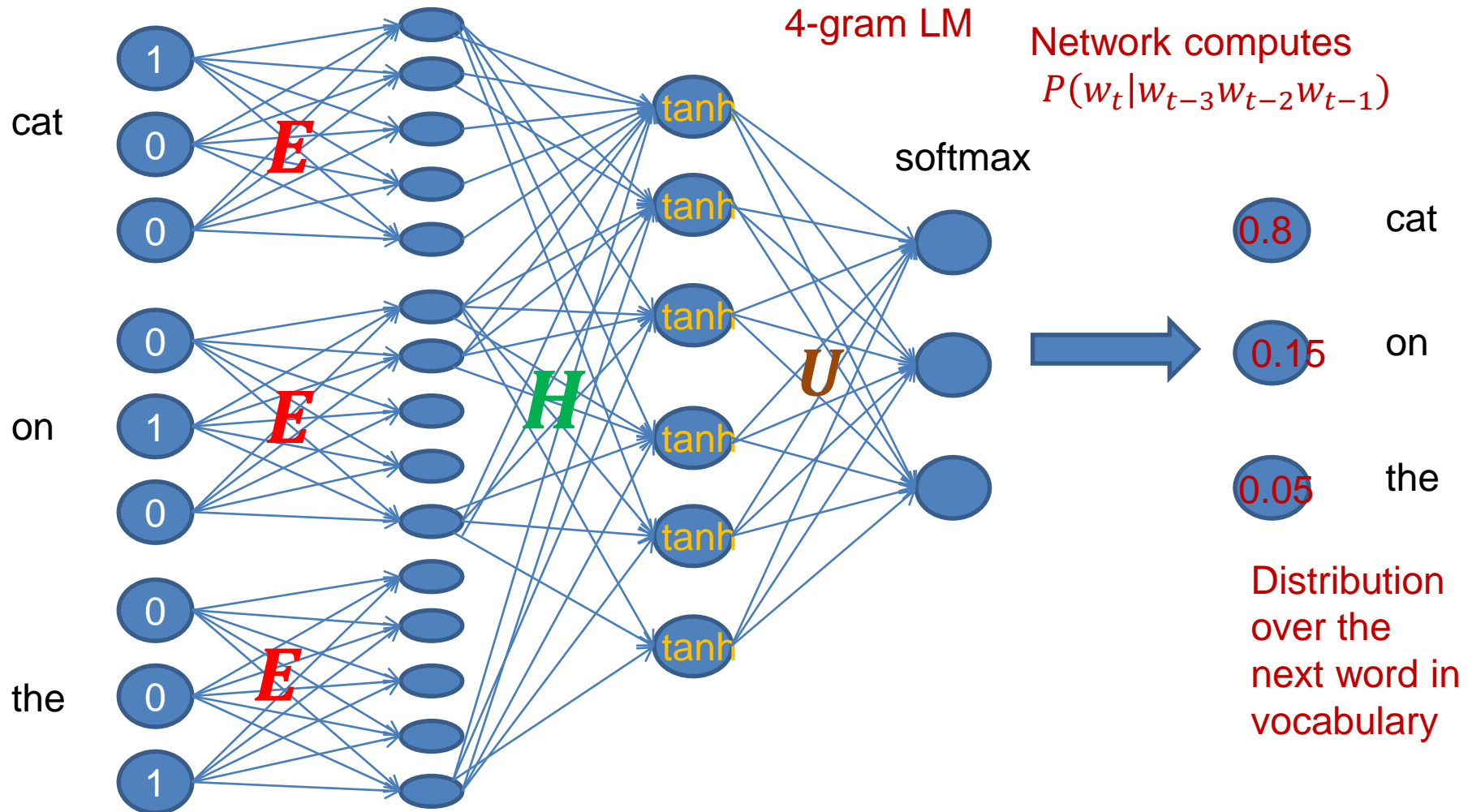
$$P(w_1 w_2 w_3 w_4 \dots) \approx P(w_1) P(w_2 | w_1) P(w_3 | w_2) P(w_4 | w_3) \dots$$

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

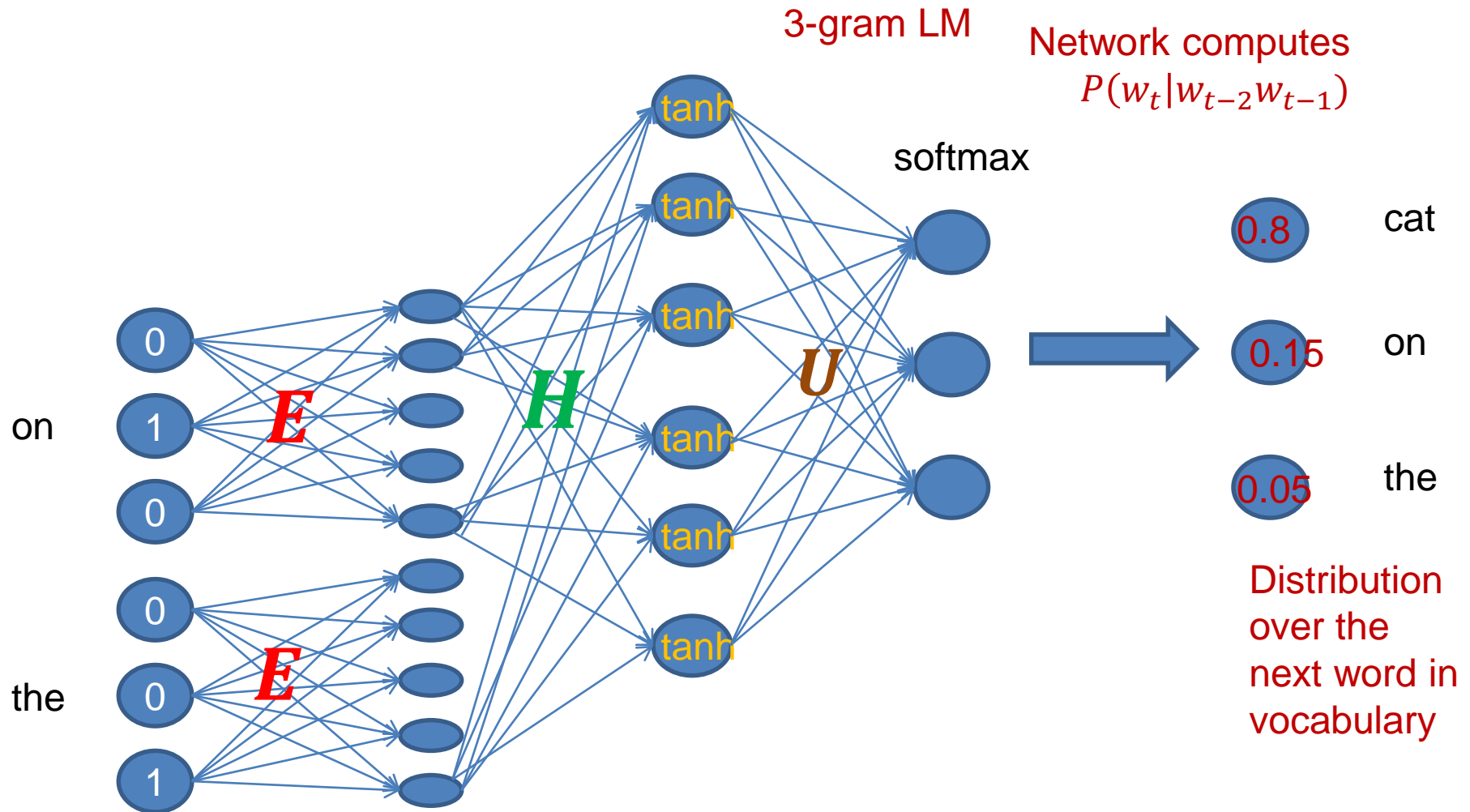
$$P(\text{dog}|\text{the}) = \frac{\text{cnt}(\text{the}, \text{dog})}{\text{cnt}(\text{the})}$$

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

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



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



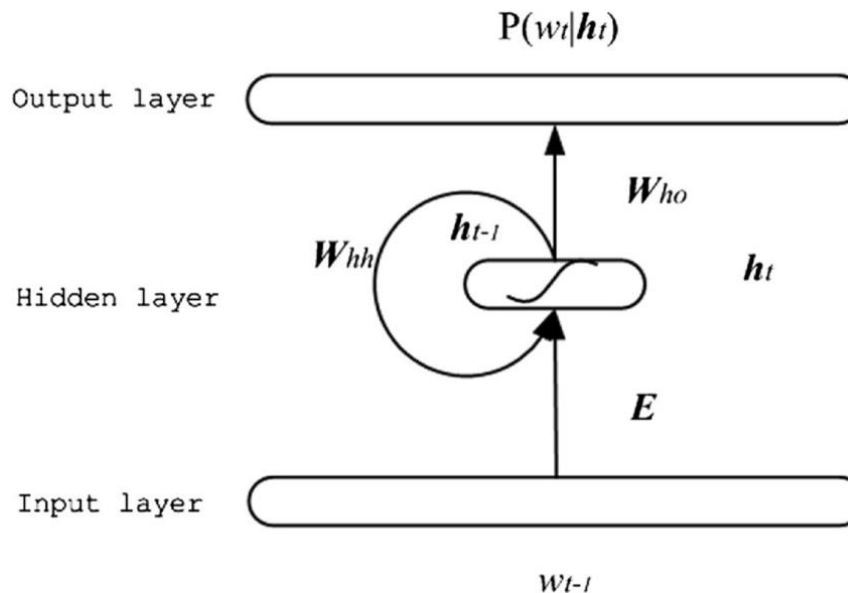
Language Models can (also) be used to Generate Language

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

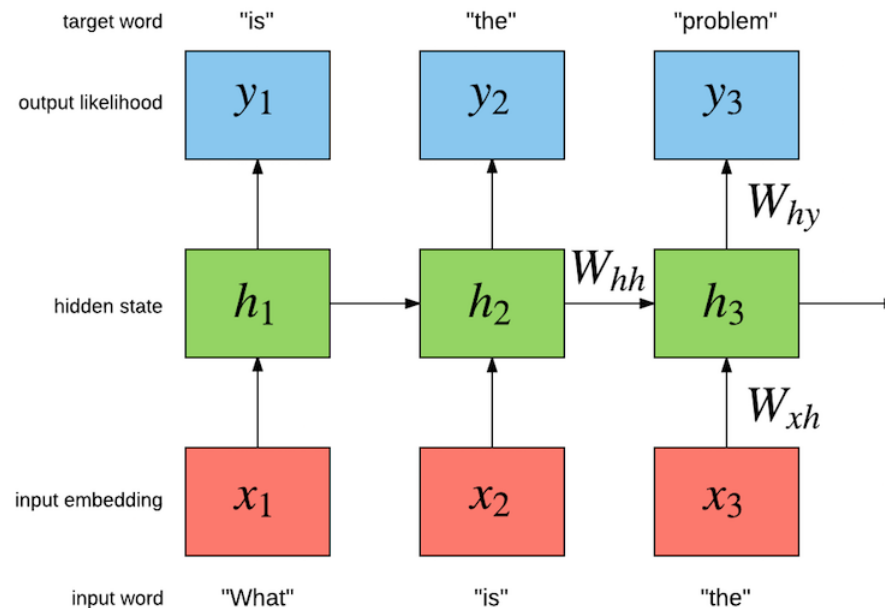
$$\begin{aligned} P(w_1 w_2 w_3 w_4 \dots) \\ \approx P(w_1) P(w_2|w_1) P(w_3|w_2) P(w_4|w_3) \dots \end{aligned}$$

- Problem with n-gram language models:
 - inherent limitations in the past window that they can take into consideration

- 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

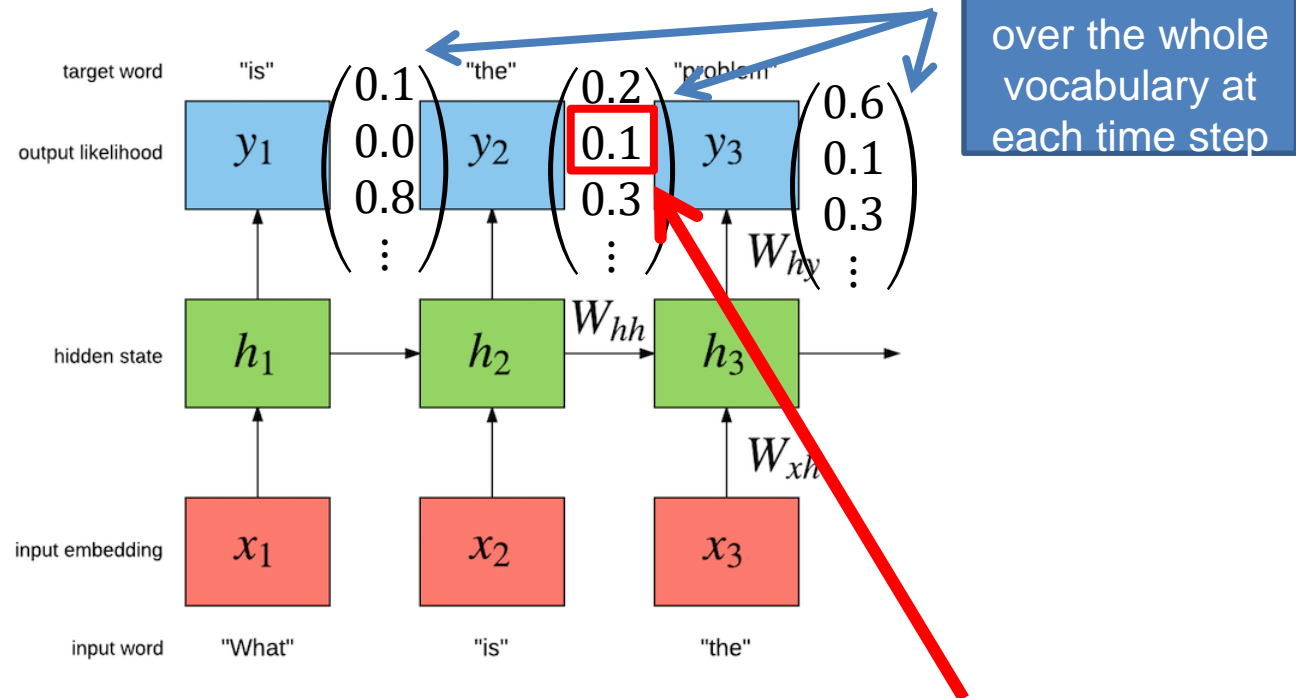


- The RNN is trained to predict the next word



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

- Example



- 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

LMs have seen a revival in NLP

- E.g., because language modeling can be a cheap **auxiliary task**
- And also due to ELMo/BERT

- 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