# OEAW AI SUMMER SCHOOL

## DEEP LEARNING II

## Neural Networks

Johannes Brandstetter, Michael Widrich
Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

# Lecture II: Neural Networks

# Perceptron

# Multi-Layer Perceptron

# Neurophysiological background

- The inside of every neuron (nerve or brain cell) carries a certain electric charge.
- Electric charges of connected neurons may raise or lower this charge:
  - by means of transmission of ions through the synaptic interface.
- As soon as the charge reaches a certain threshold, an electric impulse is transmitted through the cell's axon to the neighboring cells.
- In the synaptic interfaces, chemicals called neurotransmitters control the strength to which an impulse is transmitted from one cell to another.

# Perceptrons

- A perceptron is a simple linear threshold unit:

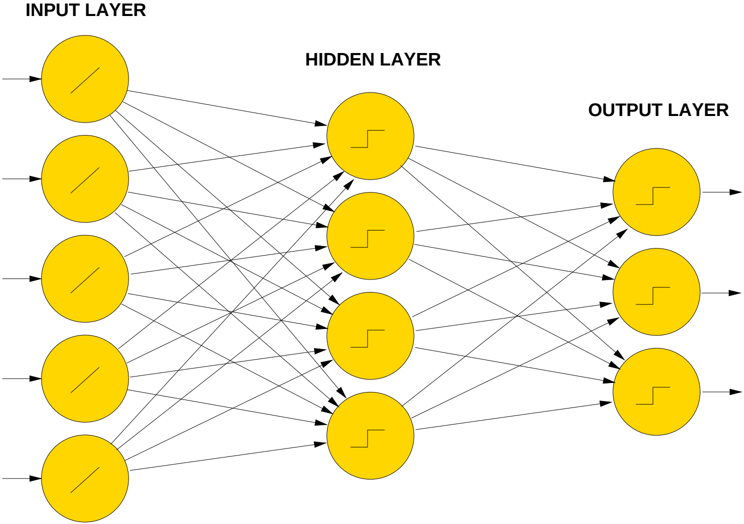
$$g(\mathbf{x}_i; \mathbf{W}, \theta) = \begin{cases} 1 & \text{if } \mathbf{W}^T \mathbf{x}_i > \theta \\ 0 & \text{otherwise} \end{cases}$$

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

# Perceptrons and linear separability

- In case that the data set $\mathbf{Z}$ is linearly separable, the perceptron learning algorithm terminates and finally solves the learning task.
- The final solution is not unique:
  - arbitrary solution (depending on initial weights)
- Perceptrons cannot solve classification tasks that are not linearly separable:
  - simple XOR problems
- Solution of introducing intermediate layers:
  - The outputs of the first layer are used as input of the first intermediate layer.

# Multi-layer perceptrons



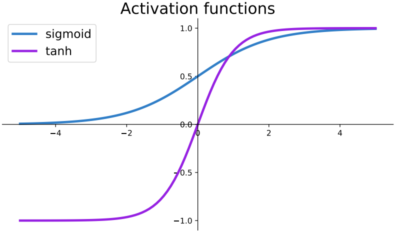
# Some historical remarks

- Minsky and Papert in the late 1960s: a training algorithm for MLPs is computationally infeasible.
  - Study of multi-layer perceptrons is not worthwhile.
  - The study of multi-layer perceptrons was almost halted until the mid of the 1980s.
- In 1986, Rumelhart, McClelland, Hinton first published the backpropagation algorithm.
  - Similarly described by Werbos (1974) and Bryson (1960s).
  - The key idea is to replace the discontinuous threshold function by a differentiable function. Then the output of the neuron, its so-called activation, is computed as:

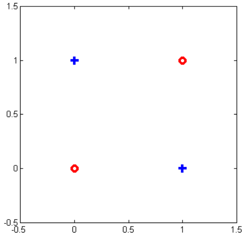
$$g(\mathbf{x}; \mathbf{W}, \theta) = \varphi\big(\mathbf{W}^T \mathbf{x} - \theta\big)$$

# Activation functions

- MLP is fully connected feed-forward neural network (FNN):
  - Today, “FNN” and “MLP” are often used synonymously.
  - Convolutional NNs are only partially connected, Recurrent NNs have feedback loops (No MLPs!).
- Without activation functions, neural networks are linear:
  - Activations allow to learn nonlinear functions.
- Earlier used activation functions: sigmoid, tanh
  - Sigmoid function $\sigma(x)$
  - $\tanh(x) = 2\sigma(2x) - 1$
  - Rescaled versions of each other



# Learning non-linear functions




| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

This classification problem:

- ... can't be solved by logistic regression.
- ... is trivial to solve using an MLP.

# How powerful are neural networks?

- Useful measure of a machine learning algorithm: what are the most complicated functions it can learn?
- Measured by the VC dimension[1]
- VC dimension for a given neural network:
  $d_{VC} \leq O(W \log(c \cdot M))$
  (W = number of weights, M = number of units)
- VC dimensions of neural networks are much higher than for e.g. Logistic Regression.
- NNs can learn (much) more complex decision functions.
- Neural Networks are "Universal Function Approximators".

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

# Universal Approximation Theorem

*For any given continuous function* $f \in I_m \equiv [0,1]^m$ *and* $\varepsilon > 0$*, there exists a function of the form*

$$F(x) = \sum_{i=1}^{N} \alpha_i \varphi \left( w_i^T x + b_i \right)$$

*such that*

$$|F(x) - f(x)| < \varepsilon$$

*for all* $x \in I_m$*.*

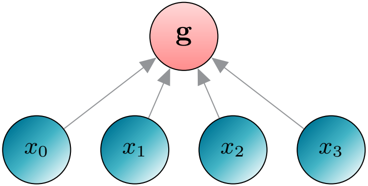
For the proof, see [Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks"]

# Backpropagation

# Vanishing Gradient

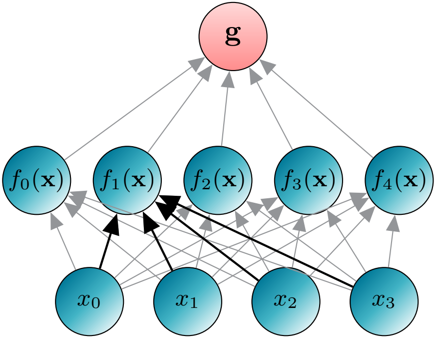
# Remember: Logistic Regression

$$g(\mathbf{x}_i) = \sigma\left(\mathbf{W}^T \mathbf{x}_i + b\right)$$




Logistic regression with **learned features**:

$$g(\mathbf{x}_i) = \sigma\left(\mathbf{W}_{(2)}^T \mathbf{f}_i + b\right)$$



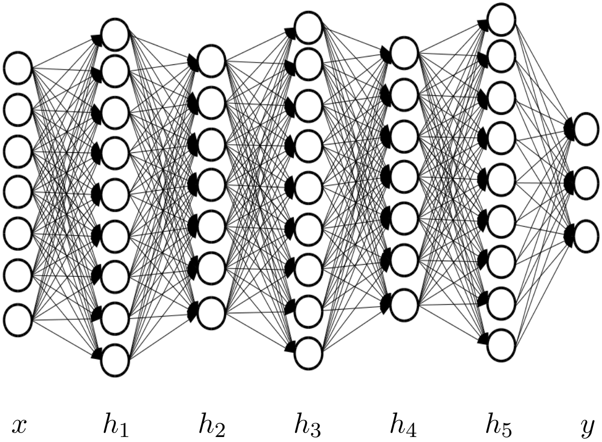
# Gradients for Logistic Regression

- **Using:** $\frac{\partial \sigma(z)}{\partial z} = \sigma(z) \cdot (1 - \sigma(z))$
- **Using** $\sigma_i$ instead of $\sigma(\mathbf{W}^T \mathbf{x}_i)$

$$
\begin{aligned}
L &= -\sum_i \left( y_i \log \sigma(\mathbf{W}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{W}^T \mathbf{x}_i)) \right) \\
\frac{\partial L}{\partial \mathbf{W}} &= -\sum_i \left( y_i \frac{1}{\sigma_i} \cdot \sigma_i \cdot (1 - \sigma_i) \cdot \mathbf{x}_i - \frac{1 - y_i}{1 - \sigma_i} \cdot \sigma_i \cdot (1 - \sigma_i) \cdot \mathbf{x}_i \right) \\
&= -\sum_i \left( y_i (1 - \sigma_i) \cdot \mathbf{x}_i - (1 - y_i) \cdot \sigma_i \cdot \mathbf{x}_i \right) \\
&= -\sum_i (y_i - y_i \sigma_i - \sigma_i + \sigma_i y_i) \mathbf{x}_i \\
&= \sum_i (\sigma_i - y_i) \mathbf{x}_i
\end{aligned}
$$

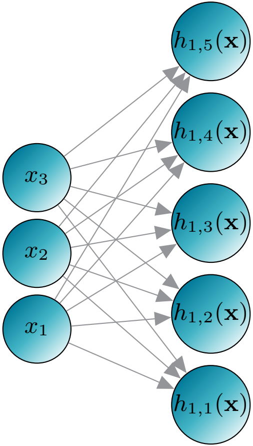
# Neural networks are nested structures

- Only fully-connected feed-forward connections:




$$y = f\Big(h_5\big(h_4\big(h_3\big(h_2\big(h_1\big(\mathbf{x}; \mathbf{W}_1\big); \mathbf{W}_2\big); \mathbf{W}_3\big); \mathbf{W}_4\big); \mathbf{W}_5\big); \mathbf{W}_6\Big)$$

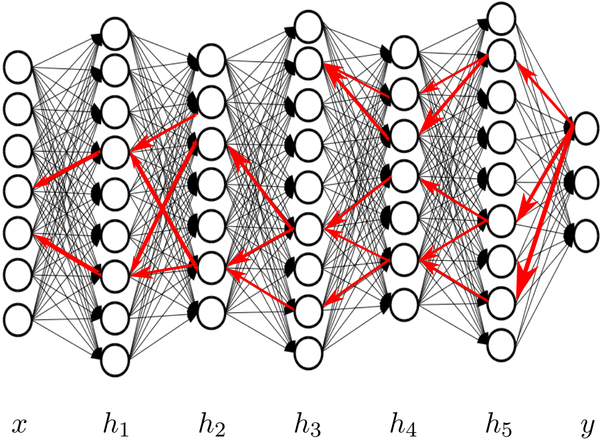
# One word how the matrices look like?




$$\underbrace{\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix}}_{\mathbf{W}_1} \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{\mathbf{x}} \xRightarrow{\text{activation}} \underbrace{\begin{pmatrix} h_{1,1} \\ h_{1,2} \\ h_{1,3} \\ h_{1,4} \\ h_{1,5} \end{pmatrix}}_{h_1}$$

# Backpropagation of errors

- Loss errors are backpropagated to update the network:




$$y = f\Big(h_5(h_4(h_3(h_2(h_1(\mathbf{x};\mathbf{W}_1);\mathbf{W}_2);\mathbf{W}_3);\mathbf{W}_4);\mathbf{W}_5);\mathbf{W}_6\Big)$$

# Backpropagation of errors

- Gradients multiply according to the chain rule.
- Gradient signal gets lost in the noise of the network:
  - **Vanishing gradient**

$$\mathbf{W}_6 \leftarrow \mathbf{W}_6 - \eta \frac{\partial L}{\partial \mathbf{W}_6}$$

$\eta$ ... learning rate

$$\mathbf{W}_5 \leftarrow \mathbf{W}_5 - \eta \frac{\partial L}{\partial h_5} \frac{\partial h_5}{\partial \mathbf{W}_5}$$

$$\mathbf{W}_4 \leftarrow \mathbf{W}_4 - \eta \frac{\partial L}{\partial h_5} \frac{\partial h_5}{\partial h_4} \frac{\partial h_4}{\partial \mathbf{W}_4}$$

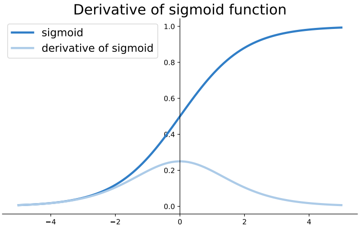
$$\mathbf{W}_3 \leftarrow \mathbf{W}_3 - \eta \frac{\partial L}{\partial h_5} \frac{\partial h_5}{\partial h_4} \frac{\partial h_4}{\partial h_3} \frac{\partial h_3}{\partial \mathbf{W}_3}$$

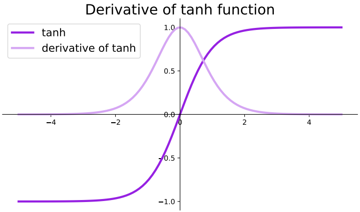
$$\mathbf{W}_2 \leftarrow \mathbf{W}_2 - \eta \frac{\partial L}{\partial h_5} \frac{\partial h_5}{\partial h_4} \frac{\partial h_4}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial \mathbf{W}_2}$$

$$\mathbf{W}_1 \leftarrow \mathbf{W}_1 - \eta \frac{\partial L}{\partial h_5} \frac{\partial h_5}{\partial h_4} \frac{\partial h_4}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial \mathbf{W}_1}$$

# Activation functions = the main culprits

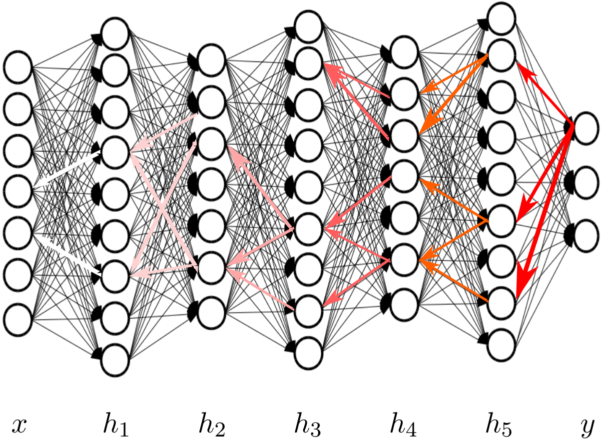
- Gradients multiply according to the chain rule.
- In the saturation regions of the activation functions the gradients are close to zero.
- First identified 1991 by Sepp Hochreiter



# Vanishing Gradient

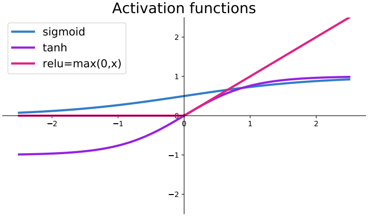
- Gradient signal gets lost in the noise:




$$y = f\left(h_5\left(h_4\left(h_3\left(h_2\left(h_1\left(\mathbf{x}; \mathbf{W}_1\right); \mathbf{W}_2\right); \mathbf{W}_3\right); \mathbf{W}_4\right); \mathbf{W}_5\right); \mathbf{W}_6\right)$$

# How to solve the vanishing gradient problem

- **Currently applied methods**:
  - Gating (mostly used in RNNs $\rightarrow$ see Lecture V)
  - Normalization: distorts gradient, increases noise
    - Batch normalization, layer normalization, weight normalization
  - Activation functions which avoid vanishing gradient
  - Clever weight initialization
  - Further regularization techniques
- Clever usage is "Trick of the Trade" in Deep Learning

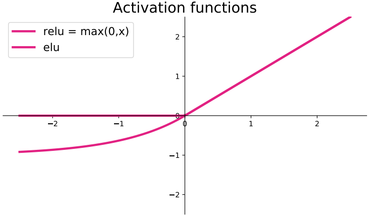


# Example: ReLUs and ELUs

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

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

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



# One word about loss functions

- **Regression**:
  - Mean-squared error for Gaussian noise assumption
  - Absolute error for Laplace noise assumption
  - If specific information about noise is known the loss function can be adapted.
- **Classification**:
  - Cross-entropy $\Rightarrow$ maximizing the likelihood of correct classification
  - Hinge loss (Support Vector Machines)

# Deep Learning

# Trick of the Trade

# Weight initialization

- 99 % of the time, a reasonably sized net will learn even with a cheap initialization.
- However:
  - Good initializations can help against Vanishing Gradients.
  - Good initializations can help to converge quickly (fewer iterations needed).
  - Good initializations are able to train even 30+ layer nets:
    - Not relevant in practice (way too deep!)
    - Interesting from research POV

# Weight initialization – simple schemes

- Simplest: $\mathbf{W}_{ij} \sim \mathcal{N}(0, 0.01)$ (or some other small $\sigma^2$)
- LeCun 1998: Make sure layer output has variance of 1:

  LeCun et al. Efficient backprop. Neural networks: Tricks of the trade.

  - Depends on activation function
  - Depends on number of input units $k$
  - Heuristic: $\mathbf{W}_{ij} \sim \mathcal{U}(\frac{-1}{k}, \frac{1}{k})$

# Weight initialization – newer schemes

- Glorot and Bengio 2010 ("Xavier initialization")

  Glorot, Bengio. Understanding the difficulty of training deep feedforward neural networks. AISTATS 2010

  - Same variance between all layers in both forward pass (activations) and backward pass (delta errors).
  - Boils down to $k \cdot \text{Var}(\mathbf{W}) = \frac{1}{3}$
  - Thus $\mathbf{W}_{ij} \sim \mathcal{U}(\frac{-\sqrt{6}}{\sqrt{k+l}}, \frac{\sqrt{6}}{\sqrt{k+l}})$:
    - $l$: number of output units in a layer
    - $k$: number of input units in a layer
    - Note: for ReLU, use $\sqrt{3}$ instead of $\sqrt{6}$

- Saxe 2014 ("orthogonal initialization")

  Saxe et al. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. ICLR 2014

  - Initialize with random orthogonal matrices
  - Reasoning comes from analyzing linear nets
  - Essential idea: keep determinant of Jacobian close to 1, then you won't lose much information

# Regularization in Deep Neural Networks

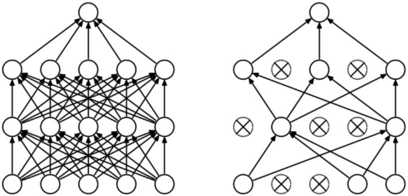
- Overfitting is an issue in neural networks.
  - Even more so in deep neural networks
- Good regularization schemes needed.
- Deep nets have enough units that each one can focus on one specific thing.
- Need to force units to prevent co-adaptation.

# Regularization in Deep Neural Networks

- Overfitting is an issue in neural networks.
  - Even more so in deep neural networks
- Good regularization schemes needed.
- Deep nets have enough units that each one can focus on one specific thing.
- Need to force units to prevent co-adaptation.
- **Presented regularization methods**:
  - Dropout
  - Batch normalization
  - Weight decay

# Dropout

- Idea: randomly "drop out" units during training
- Different units for each sample and in each iteration
- No unit can rely on the presence of other units for their work.
- Typical dropout rates: 0.5 for hidden and 0.2 for input units
- Implementation note: need to scale weights (or activations) after training



(a) Standard Neural Net (b) After applying dropout.

Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. JMLR 2014

# Batch normalization

- During learning, the output distribution of a layer will change.
  - Higher layers have to continuously adapt to this change.
- BN normalizes each minibatch to mean 0 and std 1.
- On test time, use average mean/std of training set.
- BN helps both for regularization and optimization.
- Many variants exist:
  - Weight Norm
  - Layer Norm
  - ...
- Best way to do this is still not completely understood!

# Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1 \ldots m}\}$;
Parameters to be learned: $\gamma$, $\beta$

**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

**Input:** Network $N$ with trainable parameters $\Theta$;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$

**Output:** Batch-normalized network for inference, $N_{\mathrm{BN}}^{\mathrm{inf}}$

$N_{\mathrm{BN}}^{\mathrm{tr}} \leftarrow N$ // Training BN network
**for** $k = 1 \ldots K$ **do**
  Add transformation $y^{(k)} = \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\mathrm{BN}}^{\mathrm{tr}}$ (Alg. 1)
  Modify each layer in $N_{\mathrm{BN}}^{\mathrm{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
**end for**
Train $N_{\mathrm{BN}}^{\mathrm{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
$N_{\mathrm{BN}}^{\mathrm{inf}} \leftarrow N_{\mathrm{BN}}^{\mathrm{tr}}$ // Inference BN network with frozen // parameters
**for** $k = 1 \ldots K$ **do**
  // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
  Process multiple training mini-batches $\mathcal{B}$, each of size $m$, and average over them:

$$\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

$$\mathrm{Var}[x] \leftarrow \tfrac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

  In $N_{\mathrm{BN}}^{\mathrm{inf}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$
**end for**

**Algorithm 2:** Training a Batch-Normalized Network

Ioffe, Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

ICML 2015

# Weight decay

- Weights around 0 correspond to low complexity models.
- A higher model complexity necessitates higher weights (in terms of their absolute values).
  - Therefore, a mechanism that favors weights around 0 can be used to control model complexity.
  - We add $\lambda \cdot \Omega(\mathcal{W})$ to the learning objective:
    - Regularization term $\Omega(\mathcal{W})$ measures the overall size of the weights.
    - $\mathcal{W}$ is the set of all weights in the network.
    - Regularization parameter $\lambda$ controls the influence of the regularization term.

# $L_2$ Weight Decay

- $\min_{\mathbf{W}} [\sum_{i=1}^{n} (g(\mathbf{x}_i; \mathbf{W}) - y_i)^2 + \lambda \sum_{j=1}^{m} w_j^2]$
- $\lambda = 0 \Rightarrow$ standard regression
- $\lambda = \infty \Rightarrow \mathbf{W} = 0$ (except $w_0$, straight line through mean)
- Equivalent: keep norm of $\mathbf{W}$ smaller than some given constant:

$$\min_{\mathbf{W}} \sum_{i=1}^{n} (g(\mathbf{x}_i; \mathbf{W}) - y_i)^2$$

$$\text{s. t.} \sum_{j=1}^{m} w_j^2 \leq T$$

- This form of regularization has many names:
  - $L_2$ regularization (because it penalizes the $L_2$ norm of $\mathbf{W}$)
  - Gaussian weight prior/decay
  - Ridge regression (only used for Regression, but not in e.g. neural networks)
  - Tikhonov regularization

# $L_1$ Weight Decay

- $\min_{\mathbf{W}} [\sum_{i=1}^{n} (g(\mathbf{x}_i; \mathbf{W}) - y_i)^2 + \lambda \sum_{j=1}^{m} |w_j|]$
- Penalizes $L_1$ norm of $\mathbf{W}$
- Has many names:
  - $L_1$ regularization
  - Laplace weight prior/decay
  - LASSO (least absolute shrinkage and selection operator)

    (only used for Regression, but not in e.g. neural networks)
  - Sparsity penality term
- $L_2$ Regularization $\rightarrow$ small parameters
  $L_1$ Regularization $\rightarrow$ some parameters should be put exactly to 0
- "sparse" = "contains many zeros"

# Summary

- From perceptrons to multi-layer perceptron
  - Fully connected feed forward neural networks
- Learning non-linear functions
  - Activation functions
- Backpropagation of errors
  - Vanishing gradient
- Initialization
- Regularization:
  - Dropout
  - Batch normalization
  - Weight decay