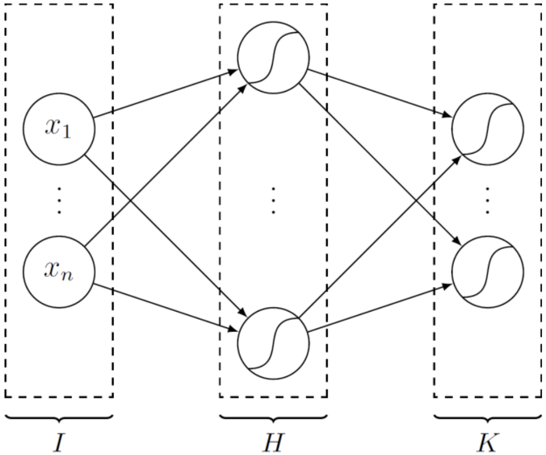
EBERHARD KARLS UNIVERSITÄT TÜBINGEN

FACULTY OF SCIENCE

Institute of Computer Science
Chair of Cognitive Modeling



Advanced Neural Networks

2. Multilayer Perceptron and Recurrent Neural Networks (Re-Visited)

Martin V. Butz
Tutoren / Co-Dozenten: Sebastian Otte

Multilayer Perceptron and Recurrent Neural Networks (Re-Visited)

Overview

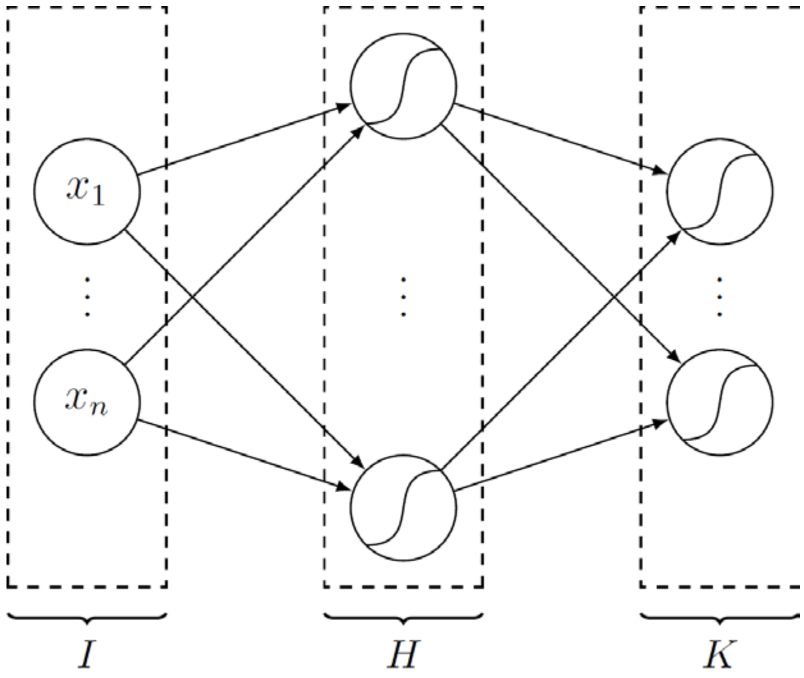
- What is a Multilayer Perceptron?
  - What does it learn?
  - How is it trained?
  - Which properties does it have?
  - How can it be manipulated?
  - How can it learn faster and more robust?

- What are Recurrent Neural Networks?
  - What (else) can they learn?
  - How are they trained?
  - What are they good for?

- Finally, next steps for understanding (Advanced) Neural Networks…

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE

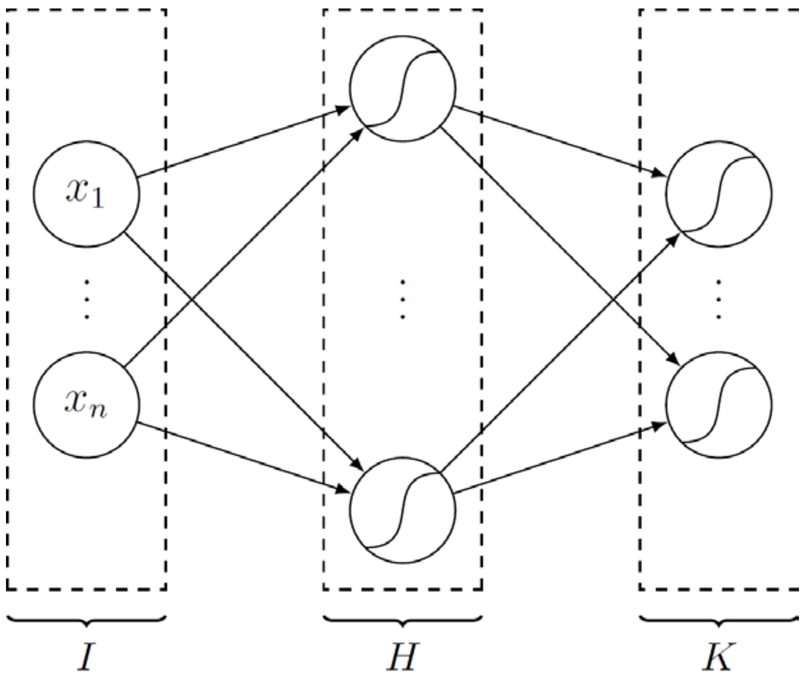


Part 1: Multilayer Perceptron

Functionality, Prediction, Error, Learning

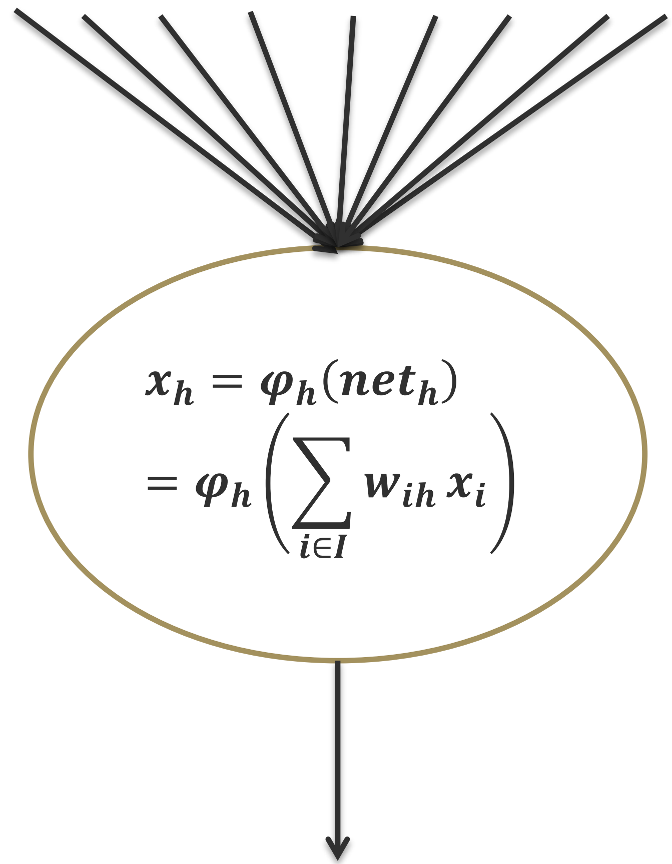
Multilayer Perceptron (MLP)

- A rather simple feed-forward neural network with
  - Input layer I
  - Hidden layer H (one or several)
  - Output layer K
- Neuron is represented by a circle.
- Within the circle, the activation function of the neuron is indicated.
  - The activation function must be differentiable.
  - The activation function in (at least some neurons of) the **hidden layers** should be non-linear.
- Dependent on the activation functions in the **output layer**, the MLP is
  - a binary (binomial) classification network,
  - a multinomial classification network, or
  - a (continuous) function approximation network.



Perceptron – One Neuron

- Simple principle to determine the current activity of a neuron *h*:
  1. Determine the weighted sum of all incoming neurons *I*.
  2. The result of the weighted sum is often denoted as net_h.
  3. Pass the result through a typically nonlinear function.
- Incoming activities may be augmented with a *bias neuron*.
- Original perceptron by Frank Rosenblatt (1928-1971) was actually a non-differentiable step function.
- Various nonlinear, differentiable activation functions are used in MLPs.
  - Sigmoid
  - Hyperbolic Tangent
  - Rectified Linear



$$x_h = \varphi_h(net_h) = \varphi_h\left(\sum_{i \in I} w_{ih} x_i\right)$$

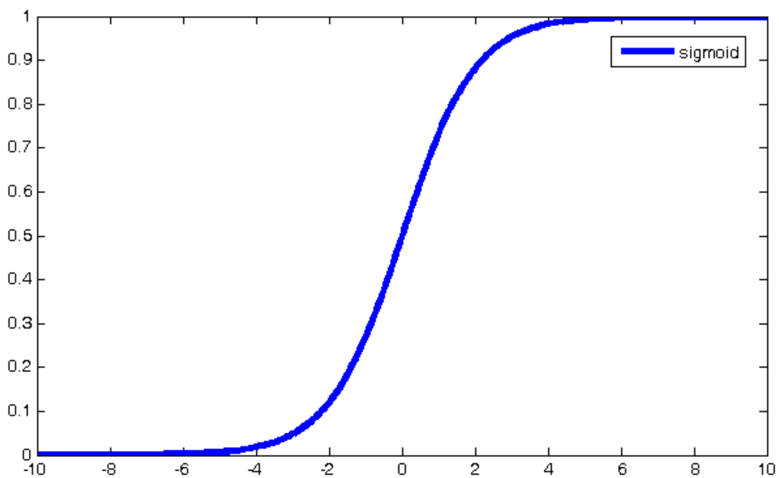
Sigmoid and Hyperbolic Tangent

Sigmoid

$$\mathrm{sig}(x) =_{\mathrm{def}} \frac{1}{1 + e^{-x}}$$

$$\mathrm{sig}(x) =_{\mathrm{def}} \frac{1}{1 + e^{-ax}}$$

$$\mathrm{sig}'(x) = \mathrm{sig}(x)(1 - \mathrm{sig}(x))$$

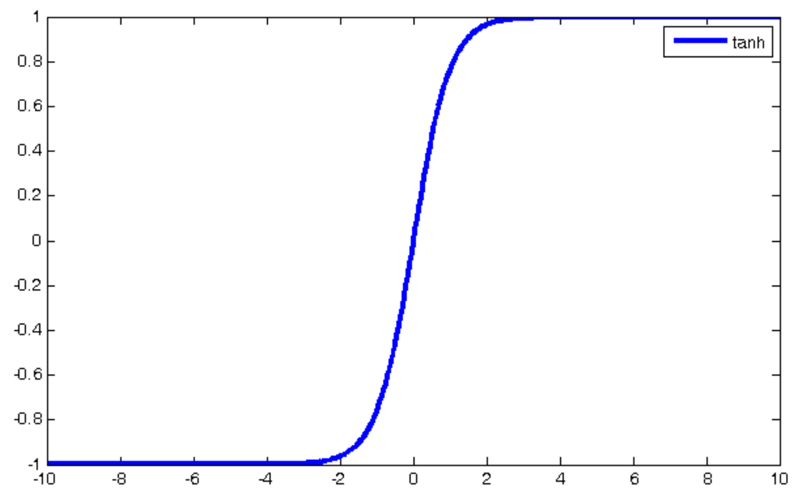


Hyperbolic Tangent

$$\tanh(x) =_{\mathrm{def}} \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$\tanh(x) =_{\mathrm{def}} \frac{e^{2ax} - 1}{e^{2ax} + 1}$$

$$\tanh'(x) = 1 - \tanh^2(x)$$

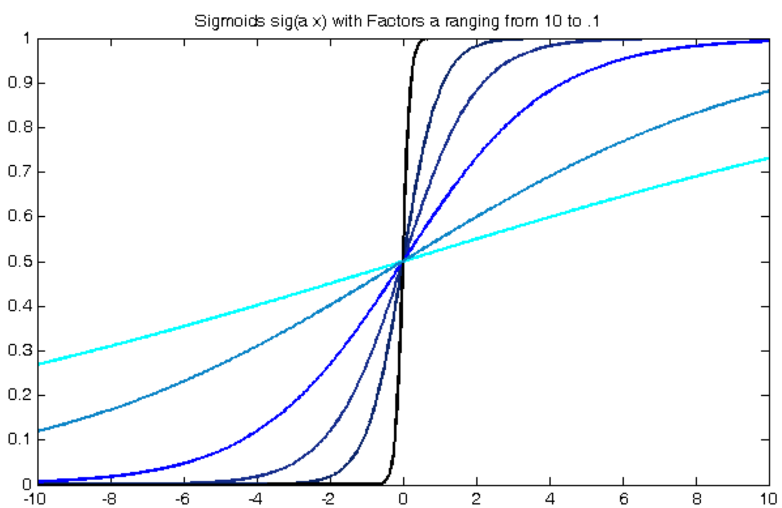


Sigmoid and Hyperbolic Tangent II

Sigmoid

$$\mathrm{sig}(x) =_{\mathrm{def}} \frac{1}{1+e^{-ax}}$$

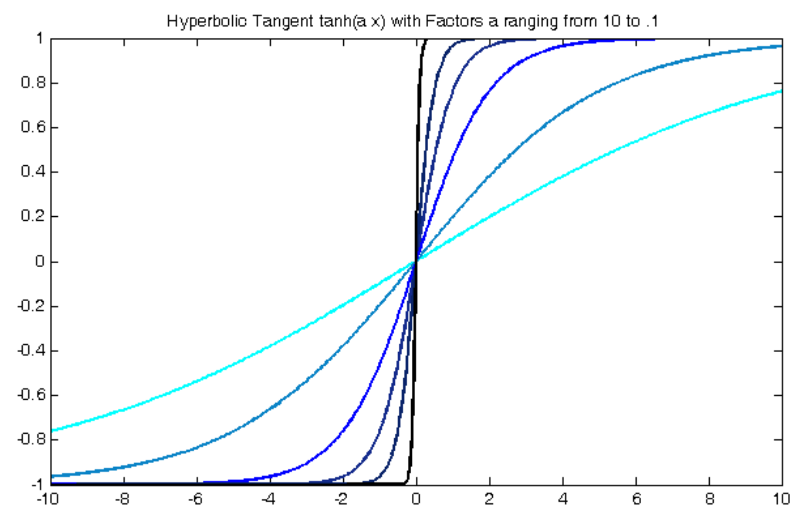
$$\mathrm{sig}'(x) = \mathrm{sig}(x)(1-\mathrm{sig}(x))$$



Hyperbolic Tangent

$$\tanh(x) =_{\mathrm{def}} \frac{e^{2ax}-1}{e^{2ax}+1}$$

$$\tanh'(x) = 1-\tanh^2(x)$$



Rectified Linear Units (ReLUs)

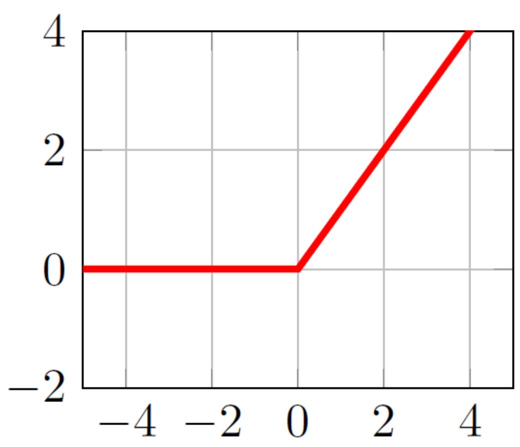
- Rather simple activation function:

$$f(x) := \begin{cases} x & \text{if } x > 0 \\ 0 & otherwise \end{cases}$$

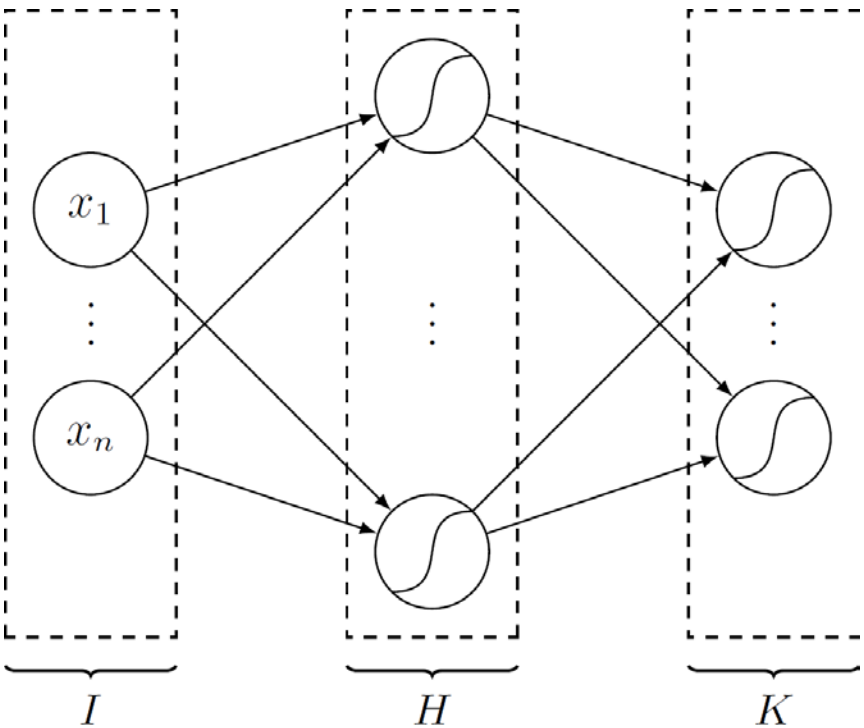
- Rather simple derivative:

$$f'(x) := \begin{cases} 1 & \text{if } x > 0 \\ 0 & otherwise \end{cases}$$

- ReLUs cause a great convergence speedup compared to sigmoidal units (up to 6 times)
- No vanishing gradient because of the linear, non-saturating shape.
- Very efficient computation, just simple operations
- HOWEVER, ReLUs have also disadvantages:
  - imagine modeling smoothly, curved shapes (curved boundaries)
  - Saturation can have positive effects in RNNs (stabilizing the dynamics)
  - Activity growth on one half-plain increases infinitely into undefined / uncovered problem subspaces.



Multilayer Perceptron Activations



$$net_h = \sum_{i \in I} w_{ih} x_i, \quad net_k = \sum_{h \in H} w_{hk} x_h,$$

$$x_h = \varphi_h(net_h). \quad x_k = \varphi_k(net_k).$$

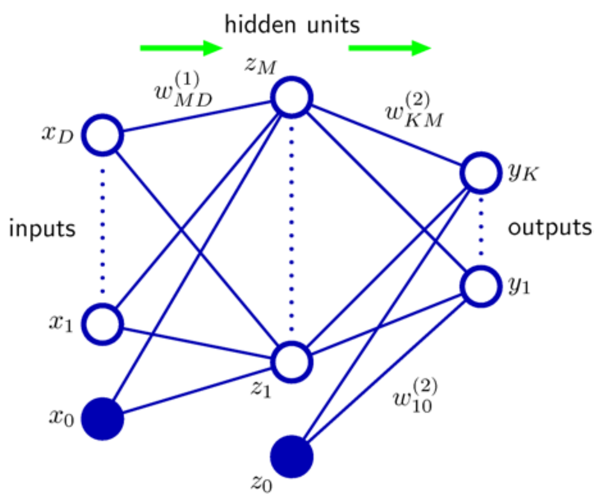
- Where
  - I is the input layer with n:=|I| neurons.
  - H is a hidden layer with |M| neurons.
  - K is the output layer with m:=|K| neurons.
- Several hidden layers are possible.
- Also direct (skip-layer) connections directly from input to output layer are possible.
- Activations of hidden and output layer neurons are determined by the weighted sum of incoming activities (net activation) passed through the activation function phi. (φ)

Note on the weight indices:
- First index = “from” index
- Second index = “to” index

Bias Neurons und Skip-Connections

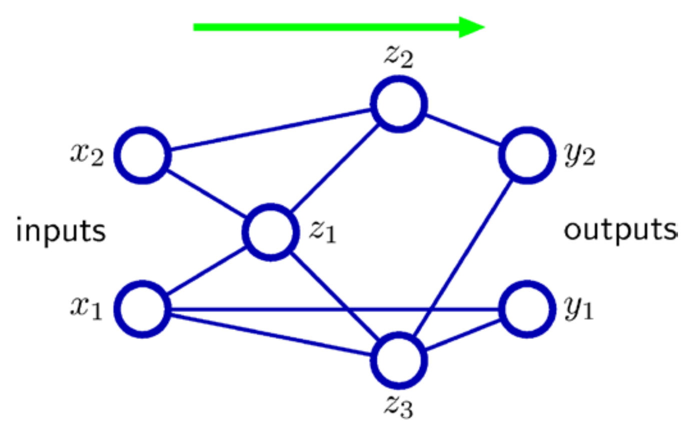
Bias Neurons

- Bias neuron (with constant activation 1) can be added to each layer.
- Effectively each activity layer (each vector) is increased by one.
- Allows constant prediction bias.



Skip Connections

- Skip connections are simply those that skip one or more hidden layer(s).
- They allow a direct influence on the output activity by the input.

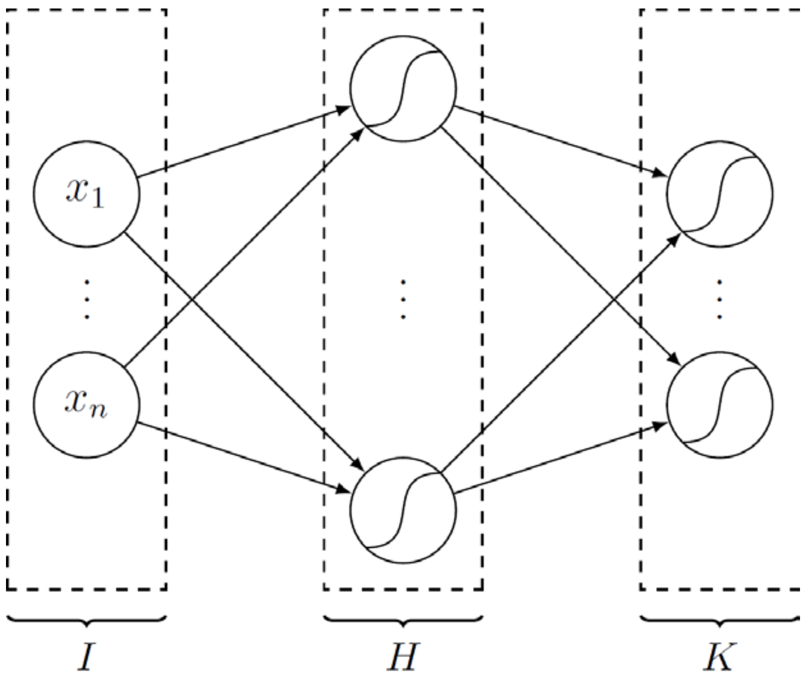


Weight-Space Symmetries

- Given the symmetric tanh activation function:
  - Note that
    tanh(-a) = - tanh(a)
  - Thus inverting all weight values into a neuron and out of a neuron results in the same network.
  - Thus, there are **$2^{|H|}$ equivalent networks**!
  - This property also holds for other activation functions.

- Note further, that one can easily exchange the incoming and outgoing connections of two hidden nodes.
  - Thus, there are **|H|! equivalent networks**.

- In total,
  - there are **$|H|!\ 2^{|H|}$ equivalent networks**!
  - A “global optimum” is
    one of $|H|!\ 2^{|H|}$ equivalent ones!



EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Objective Function

- Essentially an MLP M attempts to compute a function mapping input vector of size n onto output vector of size m. $f_M : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- The MLP computes the output by its network: $\mathbf{y} = f_M(\mathbf{x})$
- This output also depends on the network weights, thus: $\mathbf{y} = f_M(\mathbf{w}, \mathbf{x})$
- Training is then realized by means of training samples: $(\mathbf{x}, \mathbf{z})$
- The default object function is then specified by minimizing the quadratic error between a training sample and the network output:

$$E(\mathbf{z}, \mathbf{y}) =_{\text{def}} \frac{1}{2} \sum_{i=1}^{m} (z_i - y_i)^2$$

- Given a whole training set, the goal is to find those network weights that minimize the summed error:

$$\arg\min_{\mathbf{w}} \sum_{(\mathbf{x},\mathbf{z}) \in Trainset} E(\mathbf{z}, f_M(\mathbf{w}, \mathbf{x}))$$

Types of Output Layer Activations

- Binary (binomial) classification problem:
  Input belongs to one of two classes: $z \in \{0,1\}$
  - One output neuron with sigmoidal activation function.
  - Note that activation can also be considered as a probability:
    $P(z=1|\mathbf{x}) = \mathbf{y} = f(\mathbf{w},\mathbf{x})$

- Multinomial classification problem:
  Input belongs to one of several (c) classes: $z \in \{0,1,2,\ldots,c\}$
  - One output neuron with Sigmoidal activation function for each class.
  - Probability for each class $P(z=i|\mathbf{x}) = \mathbf{y_i} = f_i(\mathbf{w},\mathbf{x})$ with $i \in \{0,1,2,\ldots,c\}$

- Continuous function approximation problem:
  Input maps to a real value or vector of reals.
  - Output neurons directly map onto vector entry values.
  - Activation function is typically linear (to cover all possible values).
  - Again, a probabilistic interpretation is possible.

Cross-Entropy Error and Softmax-Activation

- For classification with neural networks the *cross-entry error* is often used as the objective function (better convergence, beneficial properties).
- In this case, the output is considered as a probabilistic (likelihood) estimate.

Binomial case (two class output, one output neuron):

- Function value *z* is 0 or 1.
- Neural output activation function is sigmoid – thus in [0,1].
- Cross-Entropy is defined as the error:

$$E(z, y) = -z \ln y - (1 - z) \ln (1 - y)$$

- Which yields the same delta error in the sigmoid case as before:

$$\delta_k = y_k - z_k$$

Multinomial case (several classes as output, one neuron for each class):

- Cross-Entropy is then:

$$E(\mathbf{z}, \mathbf{y}) = -\sum_{i=1}^{m} z_i \ln y_i$$

- Softmax activation (generalized logistic) function is used as output:

$$x_k = \frac{e^{net_k}}{\sum_{k' \in K} e^{net_{k'}}}$$

- In this case, sum of all outputs yields 1 and delta error is once again:

$$\delta_k = y_k - z_k$$

Gradient Computation

- As said: Given a whole training set, the goal is to find those network weights that minimize the summed error:

$$\arg\min_{\mathbf{w}} \sum_{(\mathbf{x},\mathbf{z})\in Trainset} E(\mathbf{z}, f_M(\mathbf{w}, \mathbf{x}))$$

- Goal: Computing the gradient with respect to the weight values!

$$\nabla_{\mathbf{w}} \left(\sum_{(\mathbf{x},\mathbf{z})\in S} E(\mathbf{z}, f_N(\mathbf{w}, \mathbf{x})) \right) = \sum_{(\mathbf{x},\mathbf{z})\in S} \nabla_{\mathbf{w}} E(\mathbf{z}, f_N(\mathbf{w}, \mathbf{x}))$$

- To minimize the error, the negative gradient should be used to adapt the weights (with a learning rate η):

$$\Delta w_{ij} =_{\text{def}} -\eta \frac{\partial E}{\partial w_{ij}}$$

Determining the Derivatives

$$\sum_{(\mathbf{x},\mathbf{z})\in S} \nabla_{\mathbf{w}} E(\mathbf{z}, f_N(\mathbf{w},\mathbf{x}))$$

$$net_k = \sum_{h\in H} w_{hk} x_h,$$

$$x_k = \varphi_k(net_k).$$

- Derivative with respect to a particular weight w_{ij} yields (via chain rule):

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

- Derivative with respect to net activation via another chain rule application:

$$\frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial x_j} \frac{\partial x_j}{\partial net_j}$$

- Where we define the delta δ_j as the derivative with respect to the net activation (which we will call the "error").

$$\delta_j =_{\text{def}} \frac{\partial E}{\partial net_j}$$

- The derivative of the net activation with respect to the weight is simply the neural activity:

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{(i',j)\in C} x_{i'} w_{i'j} = x_i$$

- Thus, the overall derivative can be written as:

$$\frac{\partial E}{\partial w_{ij}} = x_i \delta_j$$

- Finally, the derivative of the neural activation with respect to the net activation is the derivative of the activation function:

$$\frac{\partial x_j}{\partial net_j} = \frac{\partial}{\partial net_j} \varphi_j(net_j) = \varphi_j'(net_j)$$

Determining the Derivatives II:

- For the output layer, the delta error is simply the difference between network prediction x_j and function value z_j

$$\frac{\partial E}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{1}{2}\sum_{j' \in K}(z_{j'} - x_{j'})^2\right) = x_j - z_j$$

- For the hidden layer, the delta error depends on the error from the next layer.

$$\frac{\partial E}{\partial x_j} = \sum_{k \in K} \frac{\partial net_k}{\partial x_j}\frac{\partial E}{\partial net_k} = \sum_{k \in K} \frac{\partial net_k}{\partial x_j}\delta_k$$

  - Passing the error backwards through the respective weights.

$$\frac{\partial net_k}{\partial x_j} = \frac{\partial}{\partial x_j} \sum_{(j',k) \in C} w_{j'k} x_{j'} = w_{jk}$$

  - Thus, summing up the weighted error backwards through the MLP (thus, backpropagation)

$$\frac{\partial E}{\partial x_j} = \sum_{k \in K} w_{jk}\delta_k$$

Backpropagation - Procedure

1. Pass the current training example through the network, determining all net activations and resulting neural activations.

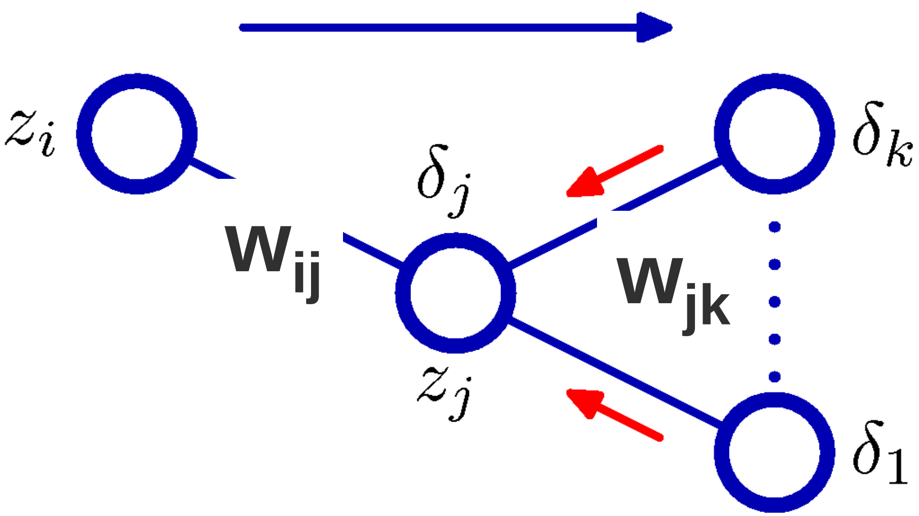
2. Determine the delta errors:
   - For each neuron k of the output layer:
     $$\delta_k = \varphi'_k(net_k)(x_k - z_k)$$
   - For each neuron h of the hidden layer:
     $$\delta_h = \varphi'_h(net_h) \sum_{k \in K} w_{hk}\delta_k$$

3. Apply weight updates by means of:
   $$\Delta w_{ij} =_{\text{def}} -\eta \frac{\partial E}{\partial w_{ij}}$$
   - With the derivatives:
     $$\frac{\partial E}{\partial w_{ij}} = x_i \delta_j$$
   - The update in time step τ can equivalently be denoted by:

$$\Delta \mathbf{w} = -\eta \nabla_{\mathbf{w}} E \qquad \mathbf{w}^{\tau} = \mathbf{w}^{\tau-1} + \Delta \mathbf{w}$$

Delta Error of Hidden Layer - Illustration

$$\delta_h = \varphi'_h(net_h) \sum_{k \in K} w_{hk} \delta_k$$



Further Notes on Backpropagation:
Weights and Learning Rate

- Weight initialization:
  - Typically uniformly random in a particular interval, such as [-0.1; 0.1]
- Learning rate η:
  - Larger learning rate:
    - faster gradient descent but
    - danger to jump over global optimum and even oscillation.
  - Small learning rate:
    - slower learning,
    - easier to get stuck in local optimum,
    - eventually more precise learning.
  - Common range of learning rate: $\eta \in [10^{-2}; 10^{-5}]$

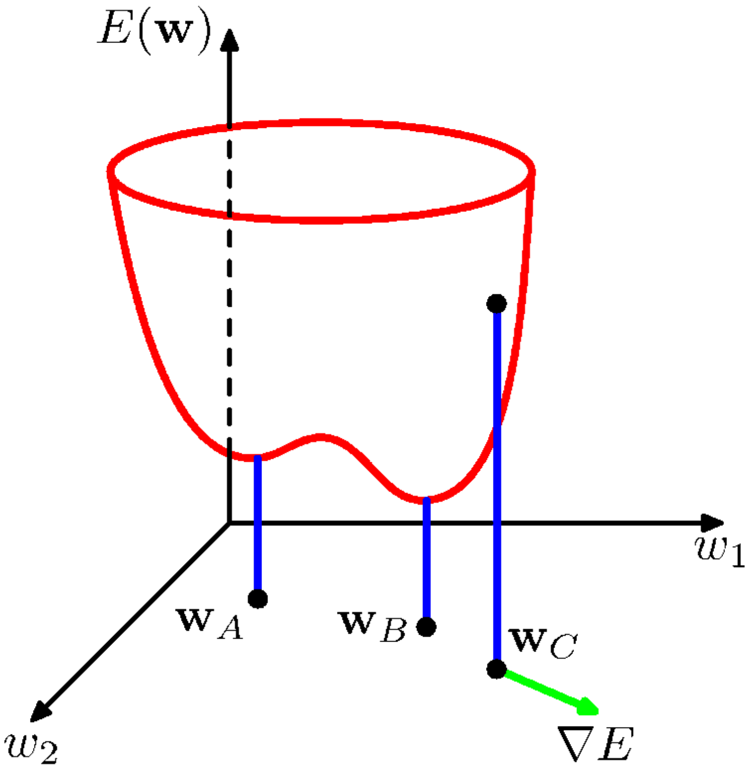
Further Notes on Backpropagation:
Online vs. Offline Learning

- An “epoch” of learning typically means to learn once from all training samples.

- Online versus offline learning:
  - Offline learning is called “batch” learning:
    - Weight update is accumulated over all samples,
    - Update is applied once per epoch.
    - Batch sampling is mathematically correct, but only practical for small training sets.
  - Online learning is also called “stochastic gradient descent”:
    - Training samples are sampled stochastically.
    - Weight update is applied after each sample.
    - Random sample order (should vary after each epoch).
    - Converges often better than batch sampling.
    - Positive effect on generalization.

Does it Work?

- Visualization of
  - Local optimum at w_A
  - Saddle point
  - Global optimum at w_B
- In practice – and especially with a sufficiently large NN – local optima are typically a minor issue.
- Remember, though, that there are many equivalent global optima ($|H|!\ 2^{|H|}$).
- Thus:
  - MLP is not guaranteed to converge to identical optimum!
  - MLP is not guaranteed to converge to global optimum (but usually does)!
  - MLP may even get stuck at a “saddle point” (very rarely).
  - MLP learning may proceed rather slowly.
- Several improvements alleviate the latter two issues to some extents:
  - Most important one: *momentum term* and more
  - Also: *stochastic gradient descent*



Important Improvements of Stochastic Gradient Descent

- **Momentum term** is used to make the weight update dependent on the previous weight update:

$$\Delta \mathbf{w}^{\tau} = -\eta \nabla_{\mathbf{w}^{\tau}} E + \mu \Delta \mathbf{w}^{\tau-1}$$

- Consequence:
  - Moving in a similar direction in the weight space over multiple weight updates.
  - Common momentum rate: $\mu = 0.9$ (while typically $\eta < 0.001$)
  - ➢ Speeds up convergence significantly.
  - ➢ Enables faster convergence even with smaller learning rates.
  - ➢ Accumulated momentum helps to jump over local minima and plateaus.
  - ➢ Induces smoothing and prevents weight value oscillations

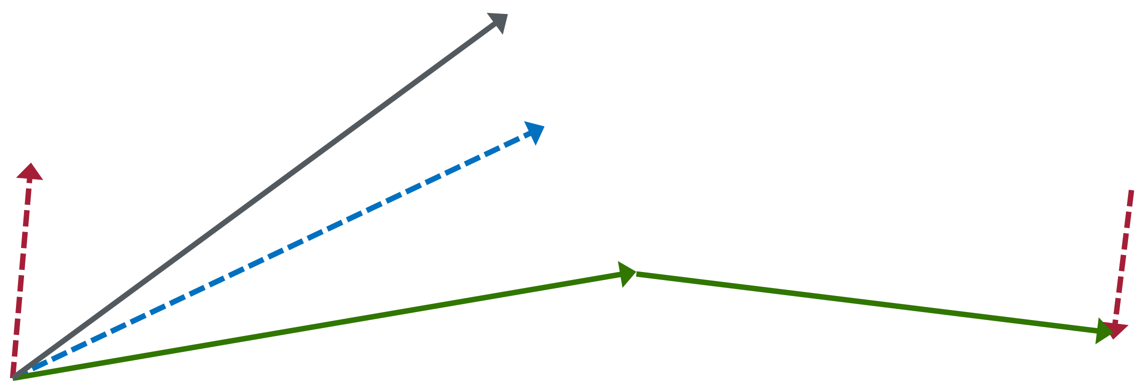
- Old but approved: Even still used in high-impact DNN/RNN papers in 2016!!!

Important Improvements of Stochastic Gradient Descent:
Nesterov Momentum

$$\Delta \mathbf{w}^{\tau} = -\eta \nabla_{\mathbf{w}^{\tau}} E\left(\mathbf{w}^{\tau} + \mu \Delta \mathbf{w}^{\tau-1}\right) + \mu \Delta \mathbf{w}^{\tau-1}$$

- First a “big” step only based on the accumulated gradient (momentum) is performed
- Second a “small” step only based on the gradient at this new point is performed

“*Its better to correct a mistake after you have made it!*”
Geoffrey Hinton

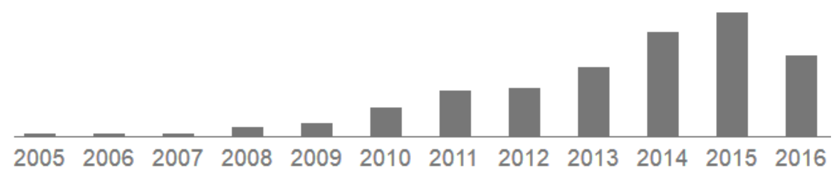


Nesterov, Y. (1983). *A method of solving a convex programming problem with convergence rate O (1/k2)*. Soviet Mathematics Doklady (Vol. 27, No. 2, pp. 372-376).
Idea used by Ilya Sutskever for ANN training in 2012

Important Improvements of Stochastic Gradient Descent:
Nesterov Momentum

$$\Delta\mathbf{w}^{\tau} = -\eta \nabla_{\mathbf{w}^{\tau}} E\left(\mathbf{w}^{\tau} + \mu\Delta\mathbf{w}^{\tau-1}\right) + \mu\Delta\mathbf{w}^{\tau-1}$$

Citations of the last decade (284 in 2015)

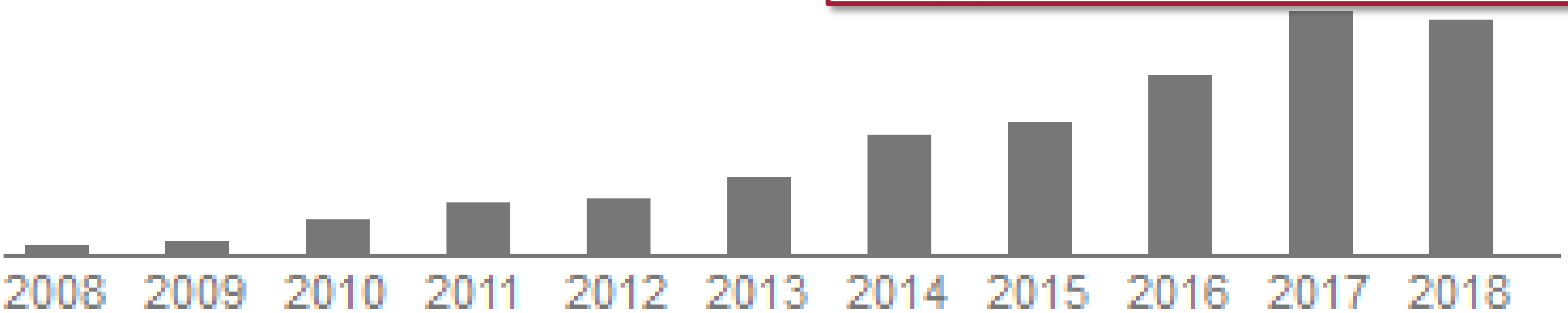


Source: google scholar

- Interesting:

Citations of the last decade (496 in 2017)

Zitiert von: 2376



Source: google scholar

Nesterov, Y. (1983). *A method of solving a convex programming problem with convergence rate O (1/k2)*. Soviet Mathematics Doklady (Vol. 27, No. 2, pp. 372-376).

Idea used by Ilya Sutskever for ANN training in 2012

RMSprop

- RMSprop: Divide the gradient by a running average of its recent squared magnitude

$$v_{ij}^{\tau} = \gamma v_{ij}^{\tau-1} + (1 - \gamma) \left[\frac{\partial E(\mathbf{w}^{\tau})}{\partial w_{ij}} \right]^2$$

"Uncentered variance"

$$\Delta w_{ij}^{\tau} = -\frac{\eta}{\sqrt{v_{ij}^{\tau} + \varepsilon}} \frac{\partial \mathcal{L}(\mathbf{w}^{\tau})}{\partial w_{ij}}$$

- Common choices: $\eta = 10^{-3}$, $\gamma = 0.9$, $\varepsilon = 10^{-8}$
- Adaptive, individual learning rate.
- RMSprop is usually better than SGD + Momentum term.

Tieleman, T. and Hinton, G. (2012). *Lecture 6.5 - RMSprop*. COURSERA: Neural Networks for Machine Learning. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Adam

- Adaptive Moment Estimation (Adam)

$$v_{ij}^{\tau} = \beta_2 v_{ij}^{\tau-1} + (1 - \beta_2)\left[\frac{\partial \mathcal{L}(\mathbf{w}^{\tau})}{\partial w_{ij}}\right]^2 \qquad \hat{m}_{ij}^{\tau} = \frac{m_{ij}^{\tau}}{1 - \beta_1^{(\tau)*}} \qquad \hat{v}_{ij}^{\tau} = \frac{v_{ij}^{\tau}}{1 - \beta_2^{(\tau)*}}$$

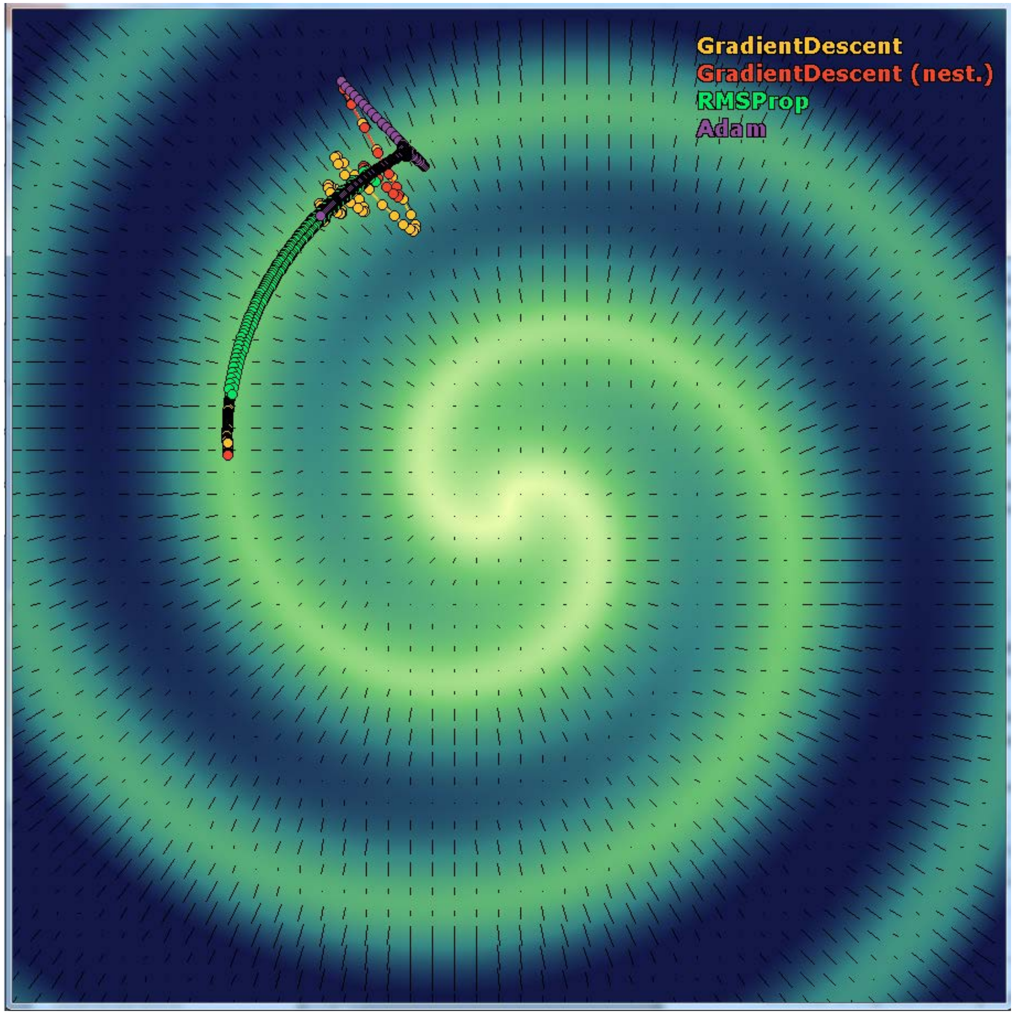
*Here (τ) is used for exponentiation

$$m_{ij}^{\tau} = \beta_1 m_{ij}^{\tau-1} + (1 - \beta_1)\frac{\partial \mathcal{L}(\mathbf{w}^{\tau})}{\partial w_{ij}} \qquad \Delta w_{ij}^{\tau} = -\frac{\eta}{\sqrt{\hat{v}_{ij}^{\tau} + \varepsilon}}\hat{m}_{ij}^{\tau}$$

- Common choices: $\eta = 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$
- Can be seen as RMSprop with smoothed gradient
- Is the current state-of-the-art

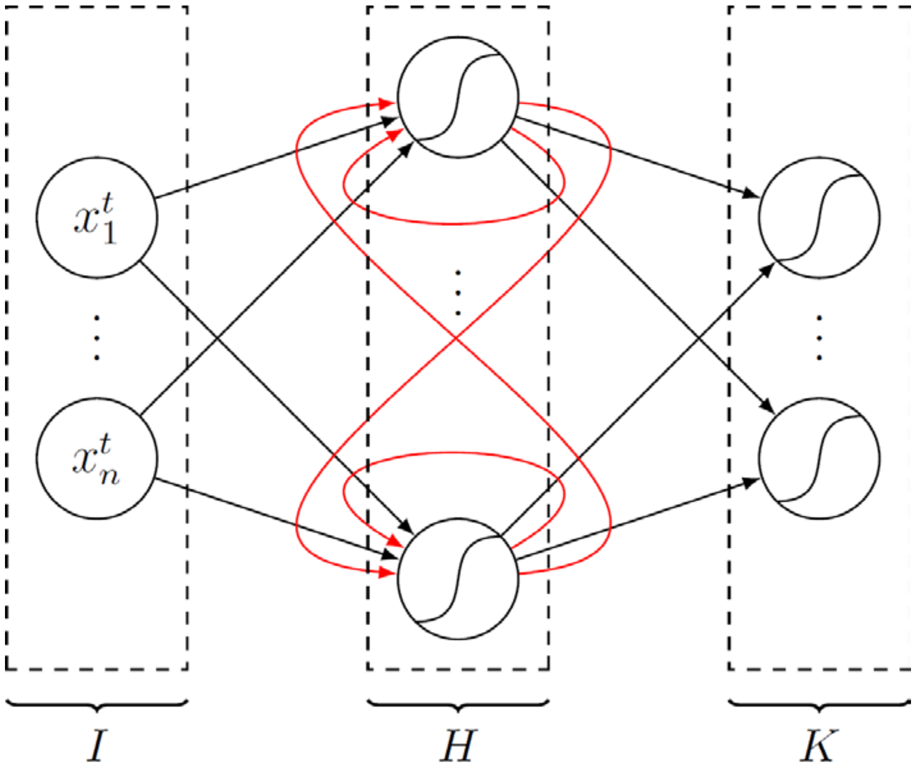
Kingma, D. P., & Ba, J. L. (2015). *Adam: a Method for Stochastic Optimization*. International Conference on Learning Representations (pp. 1-13).

Important Improvements of Stochastic Gradient Descent: Demo



EBERHARD KARLS UNIVERSITÄT TÜBINGEN

FACULTY OF SCIENCE

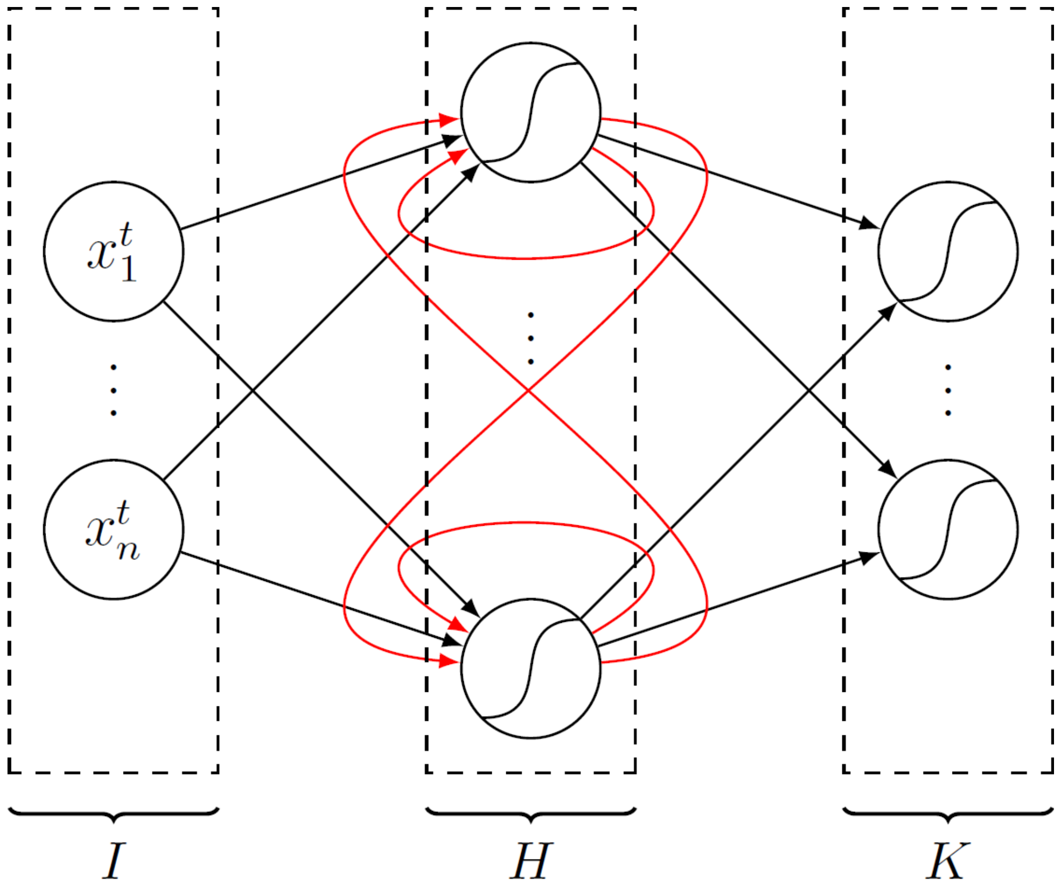


Part 2: Recurrent ANNs

Functionality, Prediction, Error, Learning

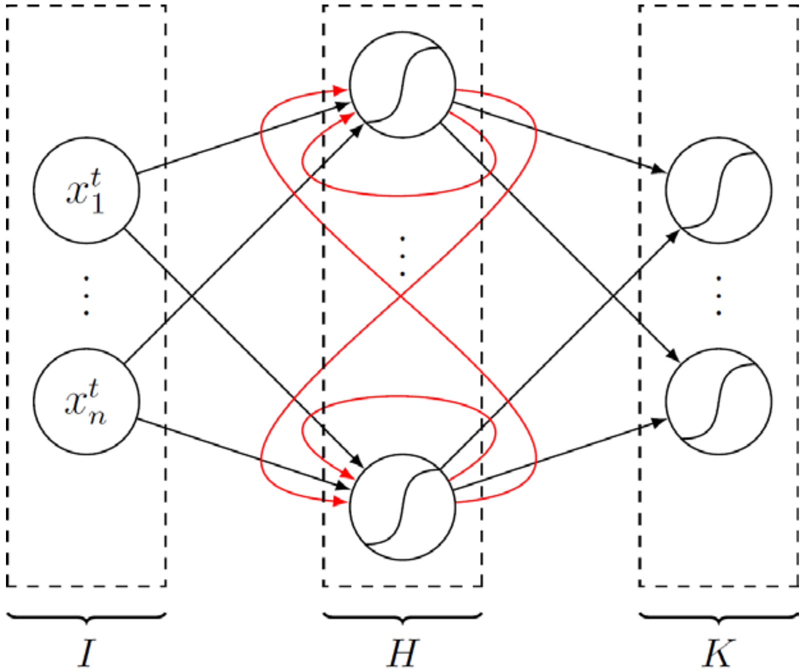
Recurrent Neural Networks

- Main difference: “recurrence”
  - Network has cycles!
- Recurrent connections can
  - “cross-”connect neurons within a layer
  - Self-connect
  - “back”-connect to a previous layer
- In order to do calculations similar to MLP, RNNs need to have time delay in recurrent connections.
- Effectively,
  - an RNN is in a certain state in each time step.
  - A state influences the state of the RNN in the next time step.
  - Thus, the same input may yield different output activities.



RNNs – Good for…

- MLP assumes "identical, independent distribution" (iid) of each data sample.
- RNN assumes dependencies between the data!
- RNN is for processing sequential data.
  - RNN processes *time series data*.
  - Often the time series is limited and RNN is trained on one full time series sample.

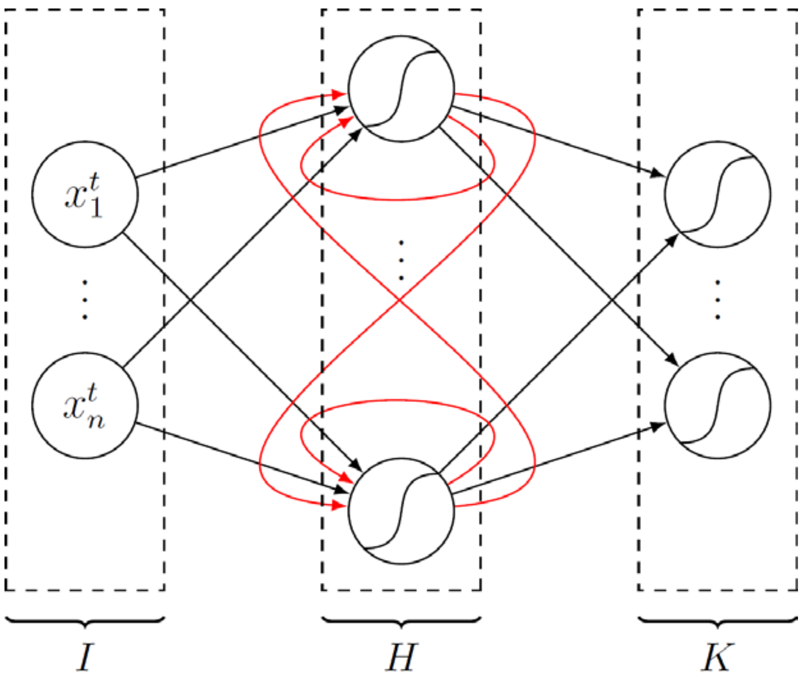


Definition of RNN

- Each neuron j with activation function φ_j
- Input vector $x^t \in \mathbb{R}^n$ at time t.
- Hidden layer activation:

$$net_h^t = \underbrace{\sum_{i \in I} w_{ih} x_i^t}_{\text{Input}} + \underbrace{\sum_{h' \in H} w_{h'h} x_{h'}^{t-1}}_{\text{Past context}}$$

$$x_h^t = \varphi_h(net_h^t)$$



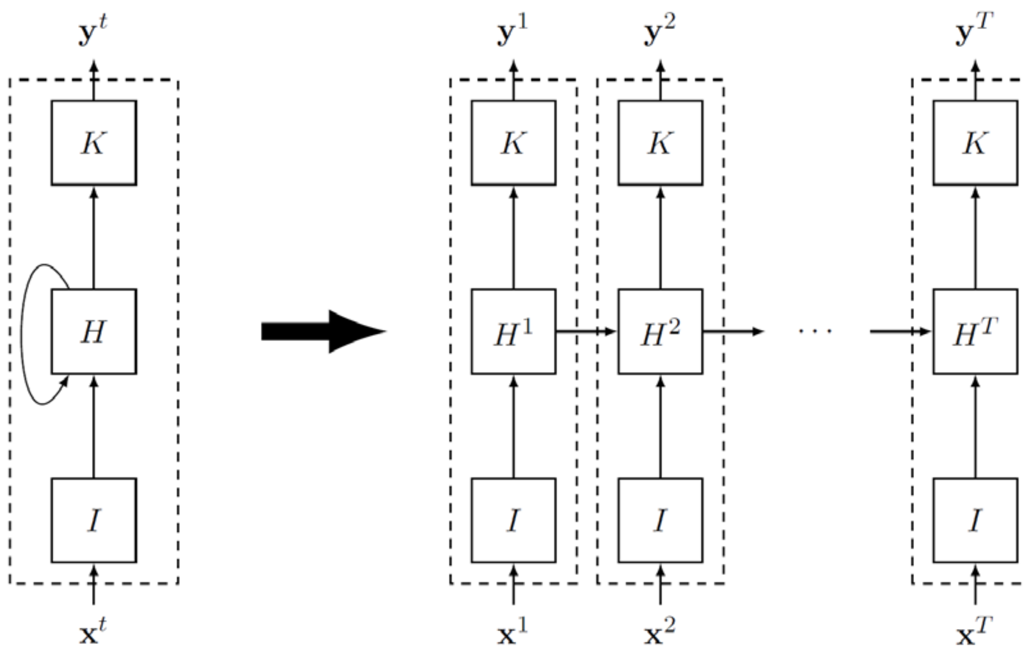
- Initialization of hidden layer activations = 0:

$$x_h^0 =_{\text{def}} 0$$

- Output layer just like in MLP.

Learning in RNNs

- In RNNs standard backpropagation cannot be used.
  - ➢ (recurrent connection would produce error loop).
- One solution in RNNs is thus to “unfold” the RNN over a (limited) number of time series iterations.
- This is called “*backpropagation through time*” (BPTT).
- Essentially, the RNN is copied multiple times T, thus producing multiple input, hidden, and output layers (one for each time step) of the same RNN.
- The recurrent connec-tions thus become forward connections in time.
- In consequence, learning rules form MLP are applicable.



Learning Rules in RNN

$$net_h^t = \underbrace{\sum_{i \in I} w_{ih} x_i^t}_{\text{Input}} + \underbrace{\sum_{h' \in H} w_{h'h} x_{h'}^{t-1}}_{\text{Past context}}$$

$$x_h^t = \varphi_h(net_h^t)$$

- Assuming a training sequence (a time series) of length T with input-output pairs $(\boldsymbol{x}^t, \boldsymbol{z}^t)$ where $t \in \{1, \ldots, T\}$
- The hidden and output layers of the RNN are unfolded in layers for each time step t ($H^1,\ldots,H^T$) and ($K^1,\ldots,K^T$).
- Note that weights stay the same:
    - $w^1_{hh'} = w^2_{hh'} = \ldots = w^T_{hh'}$
    - $w^1_{ih} = w^2_{ih} = \ldots = w^T_{ih}$
    - $w^1_{hk} = w^2_{hk} = \ldots = w^T_{hk}$
- In result, delta error computations are possible as in MLP (only difference: time index):
    - Output layer delta error: $$\delta_k^t = \varphi_k'(net_k^t)(x_k^t - z_k^t)$$
    - Hidden layer delta error: $$\delta_h^t = \varphi_h'(net_h^t)\left[\sum_{k \in K} w_{hk}\delta_k^t + \sum_{h' \in H} w_{hh'}\delta_{h'}^{t+1}\right]$$
    - Where the error from the final+1 time step is zero: $$\delta_h^{T+1} =_{\text{def}} 0$$

Learning Rules in RNN

$$net_h^t = \underbrace{\sum_{i \in I} w_{ih} x_i^t}_{\text{Input}} + \underbrace{\sum_{h' \in H} w_{h'h} x_{h'}^{t-1}}_{\text{Past context}}$$

$$x_h^t = \varphi_h(net_h^t)$$

- Output layer delta error (see previous slide):

$$\delta_k^t = \varphi'_k(net_k^t)(x_k^t - z_k^t)$$

- Hidden layer delta error (see previous slide):

$$\delta_h^t = \varphi'_h(net_h^t) \left[\sum_{k \in K} w_{hk} \delta_k^t + \sum_{h' \in H} w_{hh'} \delta_{h'}^{t+1} \right]$$

- Still open issue:
  How to compute the weight update over several time steps?
  - Remember:
    Error function E should be minimized.
  - Thus, use the
    derivative of the error as the weight update:

$$\frac{\partial E}{\partial w_{ij}} = x_i \delta_j$$

  - However, now this error extends over the whole time series sequence of length T:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{t=1}^{T} x_i^t \delta_j^t$$

Alternatives to BPTT / BPTT Handling

- Real-Time Recurrent Learning (RTRL).

- With RTRL
  - usually weights are updated after each time step
  - Error is propagated forward in time
  - RTRL requires less memory but more computation time

- Note: BPTT computes the same gradient as offline RTRL
  (weight update after the entire sequence).

- Nowadays BPTT is preferred.

- How to handle very long time series data:
  - Use windowing techniques!
    - Cut the long time series into overlapping, finite sub-time series data.
    - Apply BPTT on the time windows.

Summary

- This was an overview over
  - Multi Layer Perceptron
    (MLPs, that is, standard feed-forward neural networks),
  - Backpropagation learning,
  - Recurrent Neural Networks (RNNs), and
  - Real Time Recurrent Learning (RTRL).
- Mathematics may be slightly tedious but actually not that difficult after all!
- Remember:
  - Algorithm for MLP learning via backpropagation.
  - Stochastic Gradient Descent versus Batch learning.
  - Momentum term and even more advanced techniques.
  - RNN structure and the unfolding, in order to be able to apply BPTT.