Introduction to Neural Networks

Mathematical Tools for ITS (11MAI)

Mathematical tools, 2020

Jan Přikryl

11MAI, lecture 10

Monday, December 07, 2020

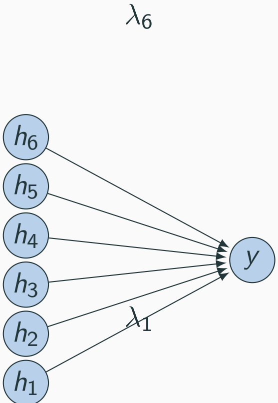
version: 2020-12-07 10:05

Department of Applied Mathematics, CTU FTS

- We used before weighted linear combination of feature values h_j and weights λ_j

$$y = \sigma(\lambda, d_i) = \sum_j \lambda_j h_j(d_i)$$

- Such models can be illustrated as a "network"

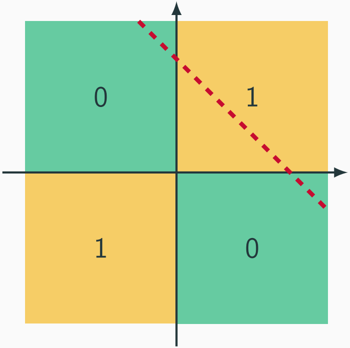


Limits of Linearity

- We can give each feature a weight
- But not more complex value relationships, e.g.,
  - any value in the range $[0; 5]$ is equally good
  - values over 8 are bad
  - higher than 10 is not worse

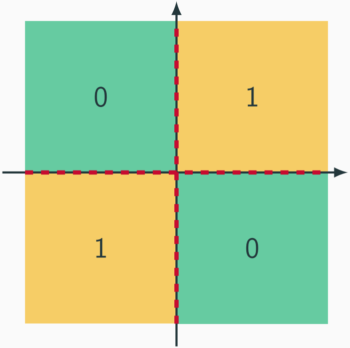
XOR

Linear models cannot model XOR-like behaviour:



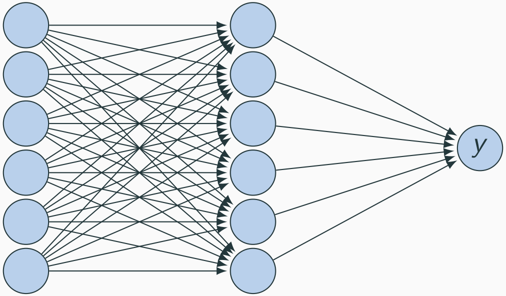
XOR

Linear models cannot model XOR-like behaviour:



Multiple Layers

Solution: Add an intermediate (“hidden”) layer of processing (each arrow is a weight)



- Have we gained anything so far?

Non-Linearity

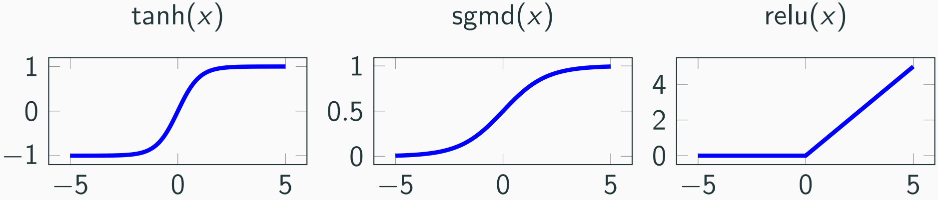
- Instead of computing a linear combination

$$\sigma(\lambda, d_i) = \sum_j \lambda_j h_j(d_i)$$

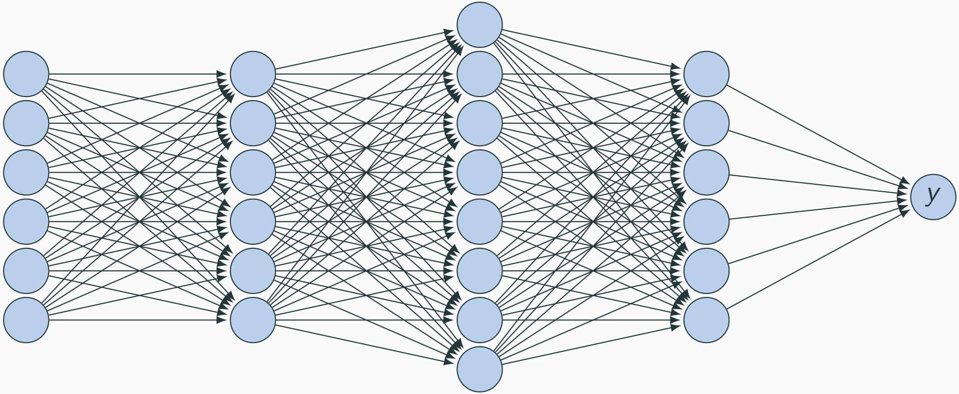
- Add a non-linear function

$$\sigma(\lambda, d_i) = f\left(\sum_j \lambda_j h_j(d_i)\right)$$

- Popular choices ("sigmoid" $\equiv$ the logistic function)



More layers = deep learning



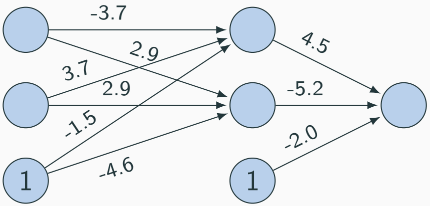
What Depths Holds

We can interpret the deep NN as follows:

- Each layer is a processing step
- Having multiple processing steps allows complex functions
- Metaphor: NN and computing circuits
  - computer = sequence of Boolean gates
  - neural computer = sequence of layers
- Deep neural networks can implement complex functions e.g. sorting on input values

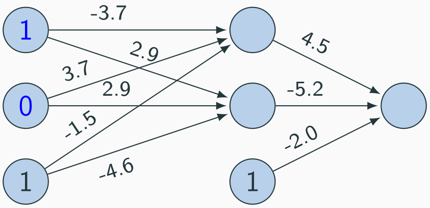
But in fact, a trained NN is just a clever lookup table.

Example



- One innovation: bias units (no inputs, always value 1)

Simple Neural Network

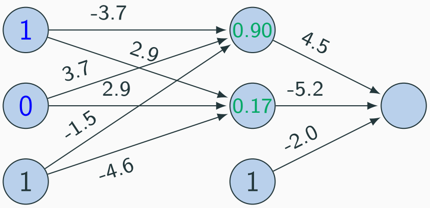


- Try out two input values
- Hidden unit computation

$$\text{sgmd}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sgmd}(2.2) = \frac{1}{1 + e{-2.2}} = 0.90$$

$$\text{sgmd}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sgmd}(-1.6) = \frac{1}{1 + e{-1.6}} = 0.17$$

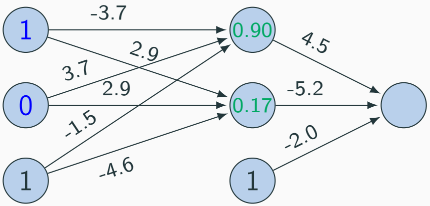
Simple Neural Network



- Try out two input values
- Hidden unit computation

$$\mathrm{sgmd}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \mathrm{sgmd}(2.2) = \frac{1}{1 + \mathrm{e}{-2.2}} = 0.90$$

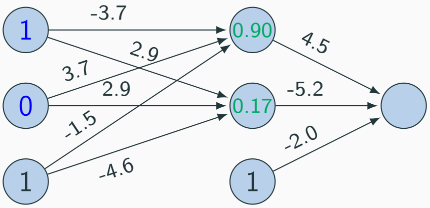
$$\mathrm{sgmd}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \mathrm{sgmd}(-1.6) = \frac{1}{1 + \mathrm{e}{-1.6}} = 0.17$$



- Output unit computation

$$\mathrm{sgmd}(0.90 \times 4.5 + 0.17 \times (-5.2) + 1 \times (-2.0)) = \mathrm{sgmd}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

Simple Neural Network



- Output unit computation

$$\mathrm{sgmd}(0.90 \times 4.5 + 0.17 \times (-5.2) + 1 \times (-2.0)) = \mathrm{sgmd}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

Output for all Binary Inputs

Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.12	0.02	$0.18 \rightarrow 0$
0	1	0.88	0.27	$0.74 \rightarrow 1$
1	0	0.73	0.12	$0.74 \rightarrow 1$
1	1	0.99	0.73	$0.33 \rightarrow 0$

- Network implements XOR
  - hidden node h_0 is OR
  - hidden node h_1 is AND
  - final layer operation is $h_0 - (-h_1)$
- Power of deep neural networks: chaining of processing steps just as: more Boolean circuits $\rightarrow$ more complex computations possible

Why “neural” networks?

Neuron in the Brain

- The human brain is made up of about 100 billion neurons

  Dendrite Axon terminal

  Soma

  Axon Nucleus
- Neurons receive electric signals at the dendrites and send them to the axon

Neural Communication

- The axon of the neuron is connected to the dendrites of many other neurons
  Neurotransmitter
  Synaptic vesicle Neurotransmitter transporter Axon terminal Voltage gated Ca++ channel
  Receptor Postsynaptic density Synaptic cleft
  Dendrite

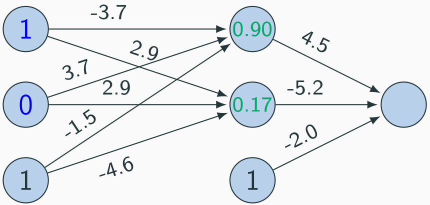
The Brain vs. Artificial Neural Networks

- Similarities
  - Neurons, connections between neurons
  - Learning = change of connections, not change of neurons
  - Massive parallel processing
- But artificial neural networks are much simpler
  - computation within neuron vastly simplified
  - discrete time steps
  - typically some form of supervised learning with massive number of stimuli

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

back-propagation training

Error



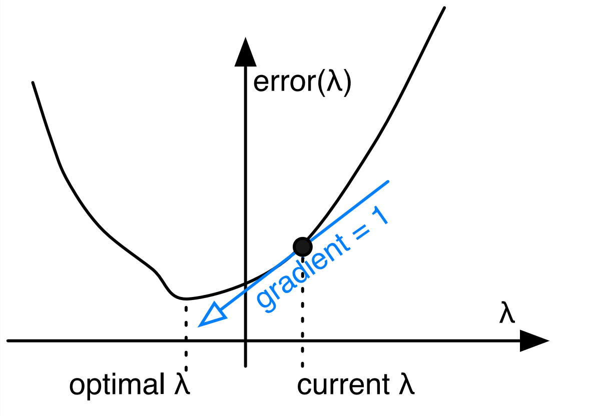
The output is not exact

- Computed output: $y = 0.76$
- Correct output: $t = 1.0$
- How do we adjust the weights?

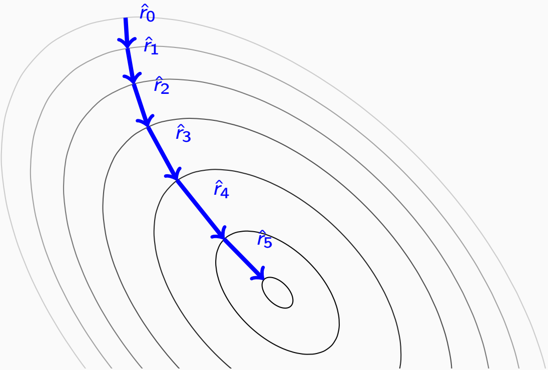
Key Concepts

- Gradient descent
  - error is a function of the weights
  - we want to reduce the error
  - gradient descent: move towards the error minimum
  - compute gradient $\rightarrow$ get direction to the error minimum
  - adjust weights towards direction of lower error
- Back-propagation
  - first adjust last set of weights
  - propagate error back to each previous layer
  - adjust their weights

Gradient Descent



Gradient Descent



Derivative of Sigmoid

- Sigmoid function: $\operatorname{sgmd}(x) = \dfrac{1}{1 + \mathrm{e}^{-x}}$
- Derivative

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}x} \operatorname{sgmd}(x) &= \frac{\mathrm{d}}{\mathrm{d}x} \frac{1}{1 + \mathrm{e}^{-x}} \\
&= \frac{(1 - \mathrm{e}^{-x}) \times 0 - 1 \times (-\mathrm{e}^{-x})}{(1 + \mathrm{e}^{-x})^2} \\
&= \frac{1}{1 + \mathrm{e}^{-x}} \cdot \frac{\mathrm{e}^{-x}}{1 + \mathrm{e}^{-x}} \\
&= \frac{1}{1 + \mathrm{e}^{-x}} \left(1 - \frac{1}{1 + \mathrm{e}^{-x}} \right) \\
&= \operatorname{sgmd}(x) \left(1 - \operatorname{sgmd}(x) \right)
\end{aligned}
$$

Final Layer Update (numbers here)

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \mathrm{sgmd}(s)$
- Error (L2 norm) $E = (t - y)^2/2$
- Derivative of error with regard to one weight w_k

$$\frac{\mathrm{d}E}{\mathrm{d}w_k} = \frac{\mathrm{d}E}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}s}\frac{\mathrm{d}s}{\mathrm{d}w_k}$$

Final Layer Update (numbers here)

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sgmd}(s)$
- Error (L2 norm) $E = (t - y)^2/2$
- Derivative of error with regard to one weight w_k

$$\frac{\mathrm{d}E}{\mathrm{d}w_k} = \frac{\mathrm{d}E}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}s}\frac{\mathrm{d}s}{\mathrm{d}w_k}$$

- Error E is defined with respect to y

$$\frac{\mathrm{d}E}{\mathrm{d}y} = \frac{\mathrm{d}}{\mathrm{d}y}\frac{1}{2}(t - y)^2 = -(t - y)$$

Final Layer Update (numbers here)

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \mathrm{sgmd}(s)$
- Error (L2 norm) $E = (t - y)^2/2$
- Derivative of error with regard to one weight w_k

$$\frac{\mathrm{d}E}{\mathrm{d}w_k} = \frac{\mathrm{d}E}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}s}\frac{\mathrm{d}s}{\mathrm{d}w_k}$$

- y with respect to x is $\mathrm{sgmd}(s)$:

$$\frac{\mathrm{d}y}{\mathrm{d}s} = \frac{\mathrm{d}}{\mathrm{d}s}\mathrm{sgmd}(s) = \mathrm{sgmd}(s)\,(1 - \mathrm{sgmd}(s)) = y\,(1 - y)$$

Final Layer Update (numbers here)

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \mathrm{sgmd}(s)$
- Error (L2 norm) $E = (t - y)^2/2$
- Derivative of error with regard to one weight w_k

$$\frac{\mathrm{d}E}{\mathrm{d}w_k} = \frac{\mathrm{d}E}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}s}\frac{\mathrm{d}s}{\mathrm{d}w_k}$$

- x is a weighted linear combination of hidden node values h_k

$$\frac{\mathrm{d}s}{\mathrm{d}w_k} = \frac{\mathrm{d}}{\mathrm{d}w_k}\sum_k w_k h_k = h_k$$

Putting it All Together

- Derivative of error with regard to one weight w_k

$$\frac{\mathrm{d}E}{\mathrm{d}w_k} = \frac{\mathrm{d}E}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}s}\frac{\mathrm{d}s}{\mathrm{d}w_k} = -(t-y)y(1-y)h_k$$

  - error
  - derivative of sigmoid: y'
- Weight adjustment will be scaled by a fixed learning rate μ:

$$\Delta w_k = \mu\,(t-y)\,y'\,h_k$$

Multiple Output Nodes

- Our example only had one output node
- Typically neural networks have multiple output nodes
- Error is computed over all j output nodes

$$E = \sum_j \frac{1}{2}(t_j - y_j)^2$$

- Weights $k \to j$ are adjusted according to the node they point to

$$\Delta w_{j \leftarrow k} = \mu(t_j - y_j)y'_j h_k$$

Hidden Layer Update

- In a hidden layer, we do not have a target output value
- But we can compute how much each node contributed to downstream error
- Definition of error term of each node

$$\delta_j = (t_j - y_j)y_j'$$

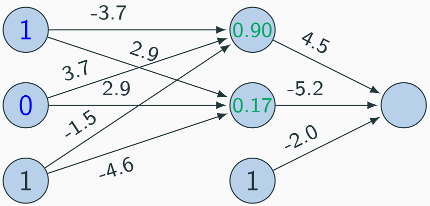
- Back-propagate the error term (why this way? there is math to back it up ...)

$$\delta_i = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y_i'$$

- Universal update formula

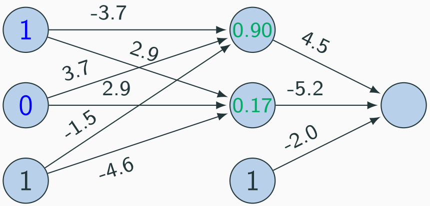
$$\Delta w_{j \leftarrow k} = \mu \delta_j h_k$$

Our Example



- Computed output: $y = 0.76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_{\mathrm{G}} = (t - y)y' = (1 - 0.76) \times 0.181 = 0.0434$
  - $\Delta w_{\mathrm{GD}} = \mu \delta_{\mathrm{G}} h_{\mathrm{D}} = 10 \times 0.0434 \times 0.90 = 0.391$
  - $\Delta w_{\mathrm{GE}} = \mu \delta_{\mathrm{G}} h_{\mathrm{E}} = 10 \times 0.0434 \times 0.17 = 0.074$
  - $\Delta w_{\mathrm{GF}} = \mu \delta_{\mathrm{G}} h_{\mathrm{F}} = 10 \times 0.0434 \times 1 = 0.434$

Hidden Layer Updates



- Hidden node **D**
  - $\delta_{\text{D}} = \left(\sum_j w_{j\leftarrow i}\delta_j\right) y'_{\text{D}} = w_{\text{GD}}\delta_{\text{G}} y'_{\text{D}} = 4.5 \times .0434 \times .0898 = .0175$
  - $\Delta w_{\text{DA}} = \mu\delta_{\text{D}} h_{\text{A}} = 10 \times 0.0175 \times 1.0 = 0.175$
  - $\Delta w_{\text{DB}} = \mu\delta_{\text{D}} h_{\text{B}} = 10 \times 0.0175 \times 0.0 = 0$
  - $\Delta w_{\text{DC}} = \mu\delta_{\text{D}} h_{\text{C}} = 10 \times 0.0175 \times 1 = 0.175$
- Hidden node **E**
  - $\delta_{\text{E}} = \left(\sum_j w_{j\leftarrow i}\delta_j\right) y'_{\text{E}} = w_{\text{GE}}\delta_{\text{G}} y'_{\text{E}} = -5.2 \times 0.0434 \times 0.2055 = -0.0464$
  - $\Delta w_{\text{EA}} = \mu\delta_{\text{E}} h_{\text{A}} = 10 \times -0.0464 \times 1.0 = -0.464$

Some additional aspects

Initialization of Weights

- Weights are initialized randomly, e.g. uniformly from interval $[-0.01, 0.01]$
- Glorot and Bengio (2010) suggest
  - for shallow neural networks

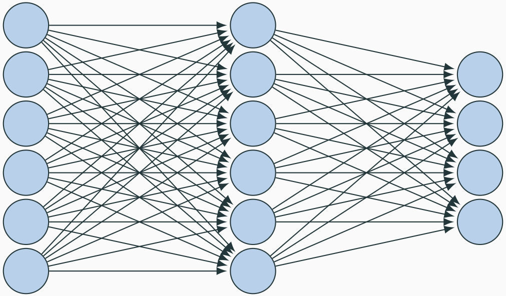
$$\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

    where n is the size of the previous layer
  - for deep neural networks

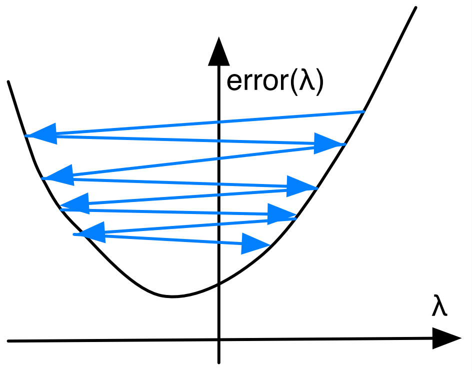
$$\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

    n_j is the size of the previous layer, n_{j+1} the size of the next layer

Neural Networks for Classification



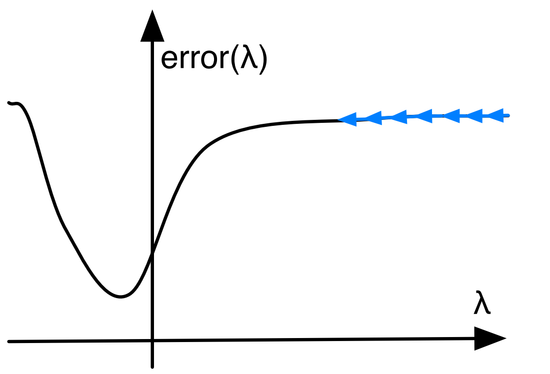
- Predict class: one output node per class
- Training data output: "One-hot vector", e.g. $\mathbf{y} = (0, 0, 1)^\mathsf{T}$
- Prediction
  - predicted class is output node y_i with highest value
  - obtain posterior probability distribution by soft-max, $\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$



Too high learning rate

Too high learning rate

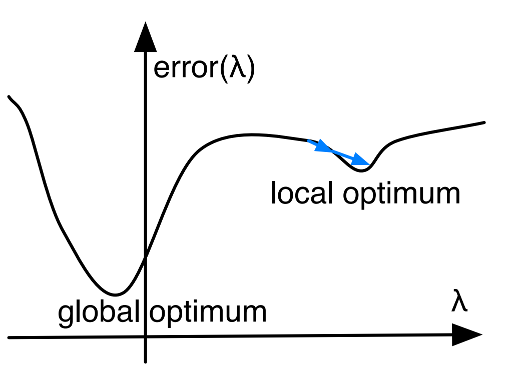
Problems with Gradient Descent Training



Bad initialization

Bad initialization

Problems with Gradient Descent Training



Local optimum

Local optimum

Speedup: Momentum Term

- Updates may move a weight slowly in one direction
- To speed this up, we can keep a memory of prior updates ...

$$\Delta w_{j \leftarrow k}(n-1)$$

- ... and add these to any new updates (with decay factor ρ)

$$\Delta w_{j \leftarrow k}(n) = \mu \delta_j h_k + \rho \Delta w_{j \leftarrow k}(n-1)$$

Adagrad

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

- Typically reduce the learning rate μ over time
  - at the beginning, things have to change a lot
  - later, just fine-tuning
- Adapting learning rate per parameter
- Adagrad update: based on error E with respect to the weight w at time $t = g_t = \frac{\mathrm{d}E}{\mathrm{d}w}$

$$\Delta w_t = \frac{\mu}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}} g_t$$

Dropout

- A general problem of machine learning: overfitting to training data (very good on train, bad on unseen test)
- Solution: *regularization*, e.g., keeping weights from having extreme values
- Dropout: randomly remove some hidden units during training
    - mask: set of hidden units dropped
    - randomly generate, say, 10–20 masks
    - alternate between the masks during training
- Why does that work? $\rightarrow$ bagging, ensemble, ...

Mini Batches

- Each training example yields a set of weight updates Δw_i.
- Batch up several training examples
  - sum up their updates
  - apply sum to model
- Mostly done or speed reasons

46

computational aspects

Vector and Matrix Multiplications

- Forward computation: $\mathbf{s} = \mathbf{W}\mathbf{h}$
- Activation function: $\mathbf{y} = \text{sgmd}(\mathbf{h})$
- Error term: $\boldsymbol{\delta} = (\mathbf{t} - \mathbf{y}) \cdot \text{sgmd}(\mathbf{s})'$
- Propagation of error term: $\boldsymbol{\delta}_i = \mathbf{W}\boldsymbol{\delta}_{i+1} \cdot \text{sgmd}(\mathbf{s})'$
- Weight updates: $\Delta\mathbf{W} = \mu\boldsymbol{\delta}\mathbf{h}^{\mathsf{T}}$

GPU

- Neural network layers may have, say, 200 nodes
- Computations such as $\mathbf{s} = \mathbf{Wh}$ require $200 \times 200 = 40\,000$ multiplications
- Graphics Processing Units (GPU) are designed for such computations
  - Real-time graphics (projections, shading) requires fast vector and matrix operations
  - GPU has massive number of multi-core but lean processing units
  - *Example:* NVIDIA Tesla K20c GPU provides 2496 thread processors, NVIDIA Tesla V100 GPU provides 5120 of them + 640 tensor cores operating on 4×4 matrices
- Extensions to C to support programming of GPUs, such as CUDA
- MATLAB is able to offload computations to GPU if parallel toolbox is installed

Toolkits

- Theano
- Tensorflow (Google) — `https://playground.tensorflow.org/`
- PyTorch (Facebook)
- MXNet (Amazon)
- DyNet

MATLAB: Deep Learning Toolbox