

Introduction to Neural Networks

Mathematical Tools for ITS (11MAI)

Mathematical tools, 2022

Jan Přikryl

(based on the book “Introduction to Statistical Learning”, <https://www.statlearning.com/>)

11MAI, lecture 9

Monday, November 28, 2022

version: 2022-11-28 12:28

Department of Applied Mathematics, CTU FTS

Neural networks became popular in the 1980s as classifiers for difficult tasks (letter and image recognition). Lots of successes, hype, and great conferences: NIPS, Snowbird.

In that time they required a lot of manual and non-intuitive tweaking to work.

Then along came SVMs, Random Forests and Boosting in the 1990s, which were simpler to set up, and Neural Networks took a back seat.

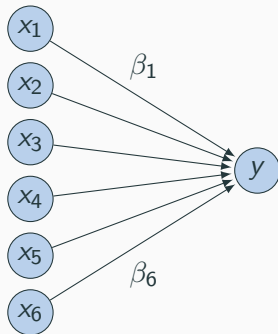
Re-emerged around 2010 as **Deep Learning**. By 2020s very dominant and successful.

Part of success due to vast improvements in computing power (GPUs), larger training sets, and free software (e.g., Tensorflow and PyTorch, plus Keras).

We have used models that can be expressed as a weighted linear combination of feature (regressor) values x_j and their weights β_j

$$y = f(\mathbf{x}) = \text{score}(\boldsymbol{\beta}, \mathbf{x}) = \sum_j \beta_j x_j$$

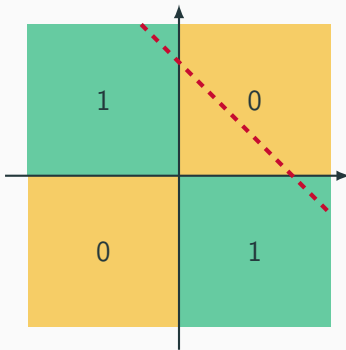
Such models can be illustrated as a “network”



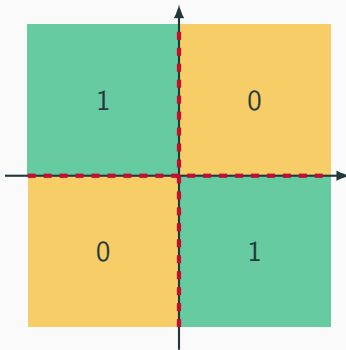
So far we have been using linear models:

- We can give each feature a weight
- But we cannot model 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
- Linear models can have **significant limitations in terms of predictive power** because the linearity assumption is almost always an approximation (and sometimes a poor one).

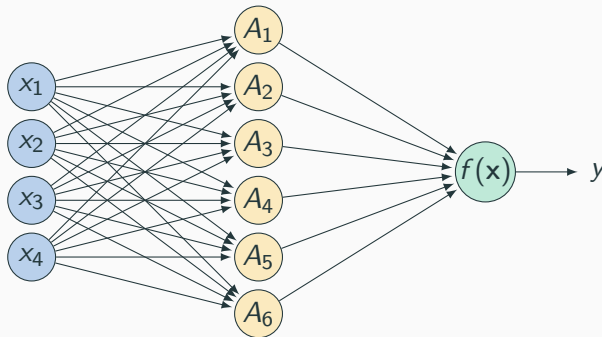
Linear models cannot model XOR-like behaviour:



Linear models cannot model XOR-like behaviour:



Solution: Add an intermediate (“hidden”) layer of processing (each arrow is a weight) and allow for neuron-inspired non-linear processing there.



$$y = f(\mathbf{x}) = \beta_0 + \sum_{k=1}^K \beta_k A_k = \beta_0 + \sum_{k=1}^K \beta_k h_k(\mathbf{x}) = \beta_0 + \sum_{k=1}^K \beta_k g(w_{k0} + \sum_{j=0}^p w_{kj} x_j)$$

The K hidden layer units process information from all p predictors:

- $A_k = h_k(\mathbf{x}) = g(w_{k0} + \sum_{j=1}^p w_{kj}x_j)$ are called the **activations** in the hidden layer.
- Non-linear function $g(z)$ is called the **activation function**.

Note:

- Activation functions in hidden layers are typically nonlinear, otherwise the model collapses to a linear model.
- The activations are like “derived features” in statistic learning, **but now they are non-linear** transformations of linear combinations of the original features.
- The model is fit by finding weights w_{pq} by minimizing residuals or computing maximum likelihood

Instead of computing a linear combination

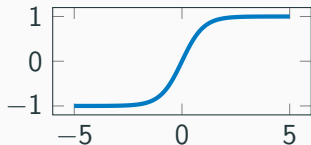
$$\text{score}(\beta, \mathbf{d}) = \beta_0 + \sum_j \beta_j h_j(\mathbf{d})$$

use a non-linear **activation function**

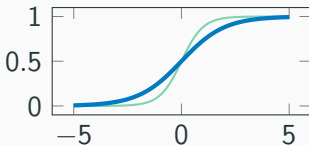
$$\text{score}(\mathbf{w}, \mathbf{d}) = g \left(w_0 + \sum_j w_j h_j(\mathbf{d}) \right)$$

Popular choices (“sigmoid” \equiv the logistic function)

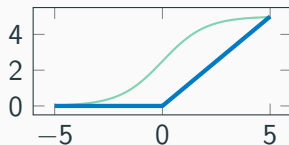
$\tanh(x)$



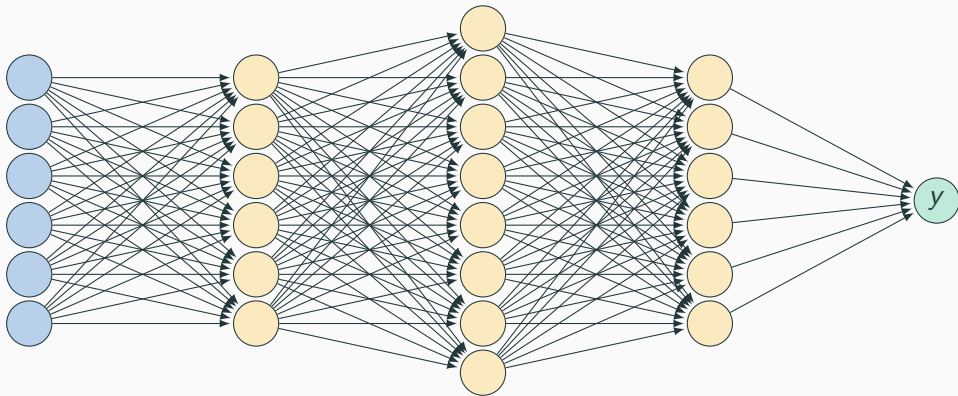
$\sigma(x)$



$\text{relu}(x)$



More layers = deep learning

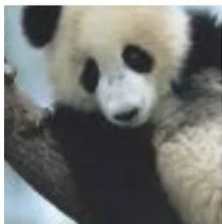


We can interpret the deep NN as follows:

- 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**.

Neural networks are often extremely sensitive to small modifications of source image:



“panda”
57.7% confidence

+

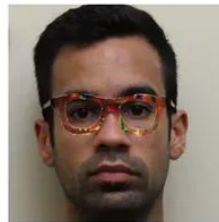


=



“gibbon”
99.3 % confidence

Neural networks are often extremely sensitive to small modifications of source image:

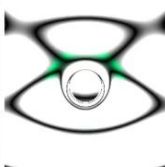


Did we say “lookup table”?

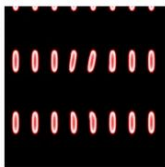
Even worse, neural networks tend to misclassify things they have not been trained on:



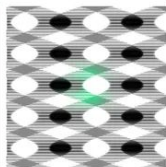
assault rifle



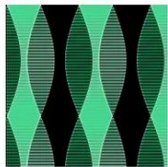
stethoscope



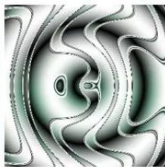
digital clock



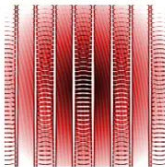
soccer ball



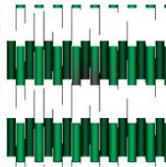
paddle



vacuum



accordion



screwdriver

Did we say “lookup table”?

Even worse, neural networks tend to misclasify things they have not been trained on:



Indian elephant



Indian elephant



African grey



African grey



common newt



carousel

What about traffic applications?

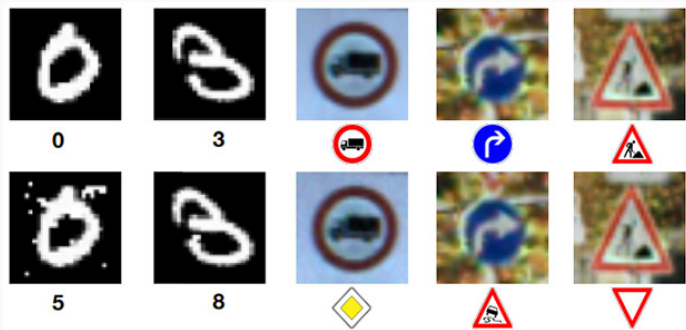
Cca 5 year ago, ANNs were not so difficult to spoof:



Everything is speed limit 45 mph!

What about traffic applications?

Cca 5 year ago, ANNs were not so difficult to spoof:



Just add a few dots here and there ...

What about traffic applications?

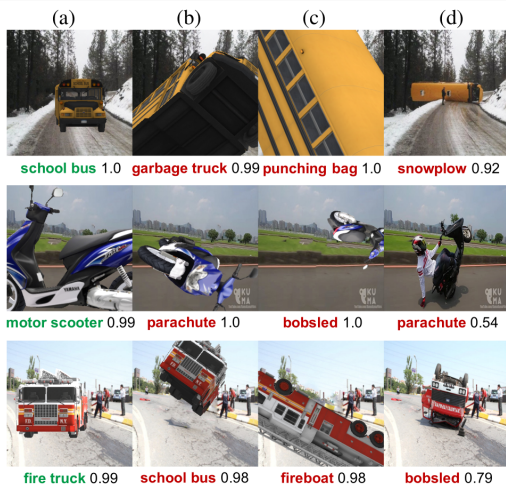
Cca 5 year ago, ANNs were not so difficult to spoof:



...or use stickers ...

What about traffic applications?

Even an incorrect pose of a familiar object may lead to unexpected results:



But it's fun anyway – Dall-E (<https://openai.com/dall-e-2/>)



Response to “medieval painting of the wifi not working”.

But it's fun anyway – Dall-E (<https://openai.com/dall-e-2/>)



Response to “Raphaele’s madonna with baby eating pizza”.

But it's fun anyway – Dall-E (<https://openai.com/dall-e-2/>)



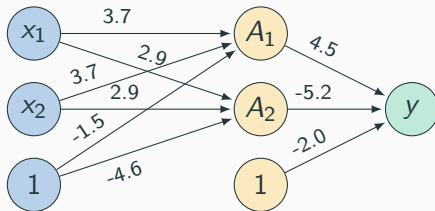
Response to “Donald Trump as the baby from the Nevermind album cover”.

But it's fun anyway – Dall-E (<https://openai.com/dall-e-2/>)

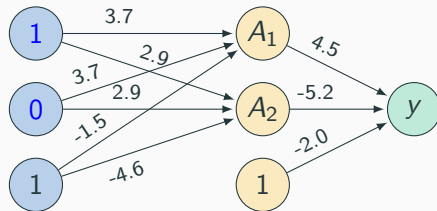


Response to “a transparent sphere on a beach with a crab looking at it”.

Example



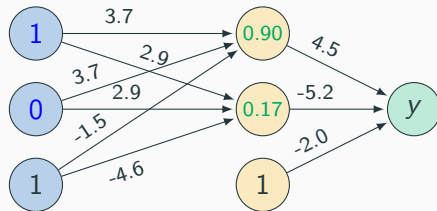
- **Bias units** (no inputs, always value 1) represent weights w_{k0}



- Try out two input values
- Hidden unit computation

$$A_1 = \sigma(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times (-1.5)) = \sigma(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

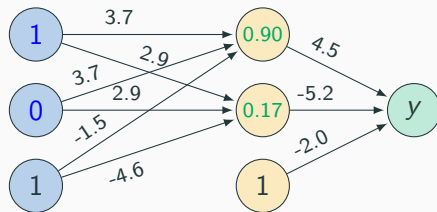
$$A_2 = \sigma(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times (-4.5)) = \sigma(-1.6) = \frac{1}{1 + e^{-1.6}} = 0.17$$



- Try out two input values
- Hidden unit computation

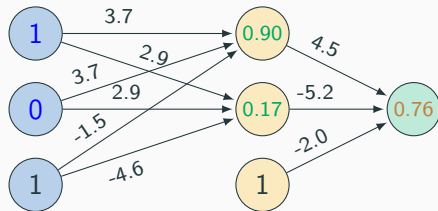
$$A_1 = \sigma(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times (-1.5)) = \sigma(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

$$A_2 = \sigma(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times (-4.5)) = \sigma(-1.6) = \frac{1}{1 + e^{-1.6}} = 0.17$$



- Output unit computation

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



- Output unit computation

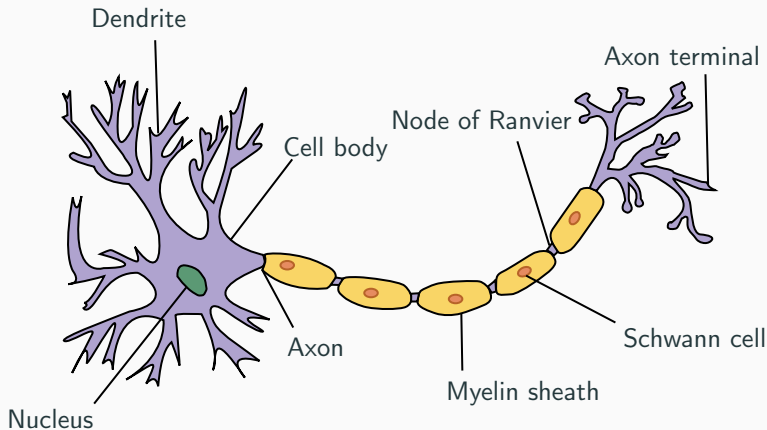
$$\sigma(0.90 \times 4.5 + 0.17 \times (-5.2) + 1 \times (-2.0)) = \sigma(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

Input x_1	Input x_2	Hidden A_1	Hidden A_2	Output y
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

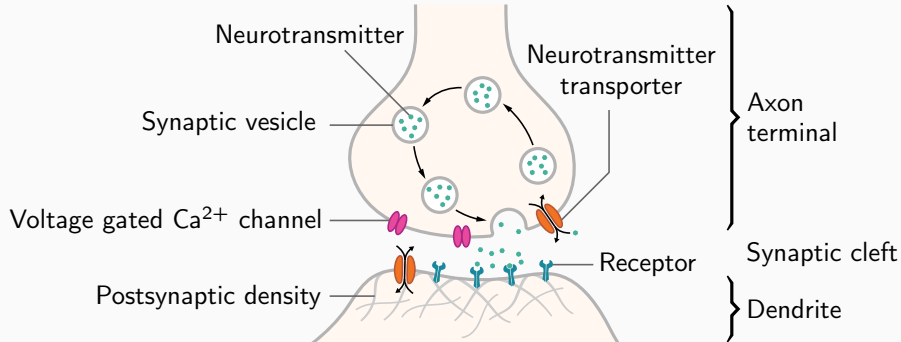
Why “neural” networks?

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

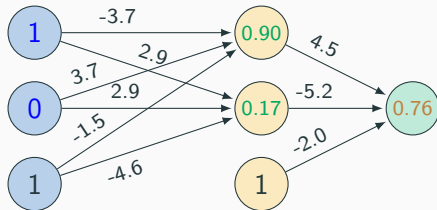


- Neurons receive electric signals at the **dendrites** and send them to the **axon**

The axon of the neuron is connected to the dendrites of many other neurons



- 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



The output is not exact

- Computed output: $\hat{y} = 0.76$
- Correct output: $y = 1.0$

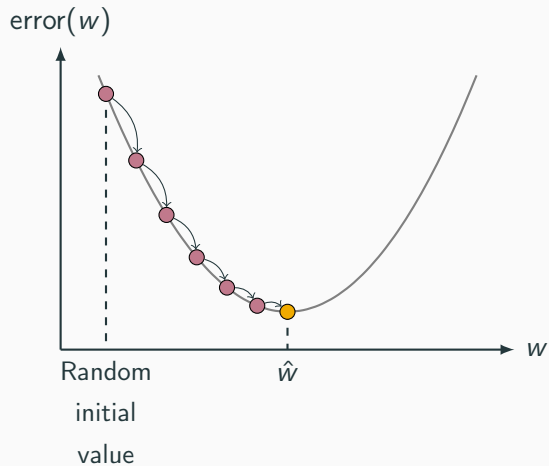
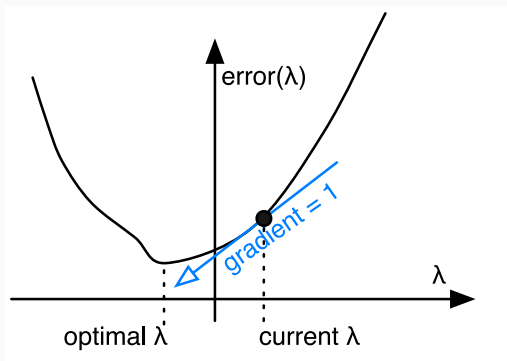
How do we correct the weights so that the error decreases?

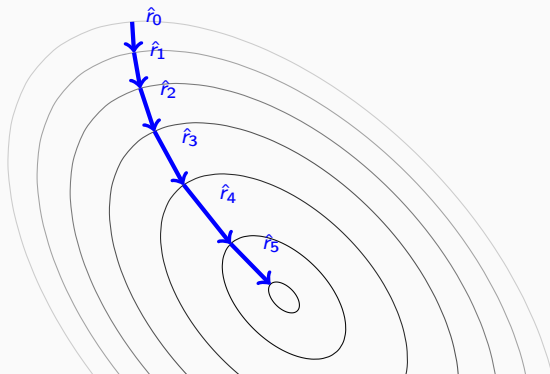
Find minimal training error using non-linear minimization, e.g. by **gradient descent**:

- error is a function of the weights
- we want to reduce the error
- move towards the error minimum
- gradient descent: compute gradient \rightarrow get direction to the error minimum
- requires (numerical) derivative of the error function

Adjust weights towards direction of lower error using **back-propagation**:

- first adjust last set of weights
- propagate error back to each previous layer
- adjust their weights





- Sigmoid function: $\sigma(x) = \frac{1}{1 + e^{-x}}$
- Derivative

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

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $\hat{y} = \sigma(s)$
- We will use L2 error norm: $E = (y - \hat{y})^2/2$
- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{d\hat{y}} \frac{d\hat{y}}{ds} \frac{ds}{dw_k}$$

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $\hat{y} = \sigma(s)$
- We will use L2 error norm: $E = (y - \hat{y})^2/2$
- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{d\hat{y}} \frac{d\hat{y}}{ds} \frac{ds}{dw_k}$$

- Error E is defined with respect to y

$$\frac{dE}{d\hat{y}} = \frac{d}{d\hat{y}} \frac{1}{2} (y - \hat{y})^2 = -(y - \hat{y})$$

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $\hat{y} = \sigma(s)$
- We will use L2 error norm: $E = (y - \hat{y})^2/2$
- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{d\hat{y}} \frac{d\hat{y}}{ds} \frac{ds}{dw_k}$$

- Activation output \hat{y} is $\sigma(s)$:

$$\frac{d\hat{y}}{ds} = \frac{d}{ds} \sigma(s) = \sigma(s) (1 - \sigma(s)) = \hat{y} (1 - \hat{y})$$

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $\hat{y} = \sigma(s)$
- We will use L2 error norm: $E = (y - \hat{y})^2/2$
- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{d\hat{y}} \frac{d\hat{y}}{ds} \frac{ds}{dw_k}$$

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

$$\frac{ds}{dw_k} = \frac{d}{dw_k} \sum_k w_k h_k = h_k$$

- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k} = -(y - \hat{y}) \hat{y} (1 - \hat{y}) h_k$$

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

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

Our example only had one output node

But neural networks may have multiple output nodes ...

- Error is computed over all j output nodes

$$E = \sum_j \frac{1}{2} (y_j - \hat{y}_j)^2$$

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

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

There is no “true” target output value for the nodes in the hidden layer, so how do we update the weights there?

- We can compute how much each node contributed to downstream error
- Definition of error term of each node

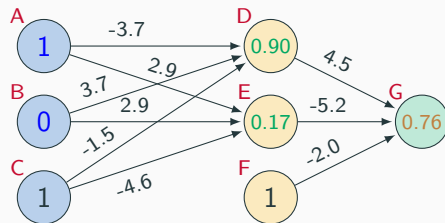
$$\delta_j = (y_j - \hat{y}_j) \hat{y}_j'$$

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

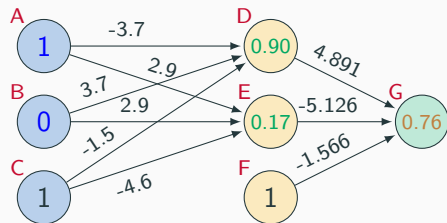
$$\delta_i = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y_i'$$

- Universal update formula

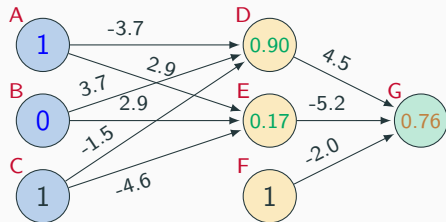
$$\Delta w_{j \leftarrow k} = \mu \delta_j h_k$$



- Computed output: $\hat{y} = 0.76$
- Correct output: $y = 1.0$
- Final layer weight updates for learning rate $\mu = 10$
 - $\delta_G = (y - \hat{y})\hat{y}' = (1 - 0.76) \times 0.181 = 0.0434$
 - $\Delta w_{GD} = \mu \delta_G h_D = 10 \times 0.0434 \times 0.90 = 0.391$
 - $\Delta w_{GE} = \mu \delta_G h_E = 10 \times 0.0434 \times 0.17 = 0.074$
 - $\Delta w_{GF} = \mu \delta_G h_F = 10 \times 0.0434 \times 1 = 0.434$

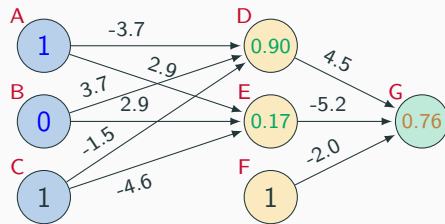


- Computed output: $\hat{y} = 0.76$
- Correct output: $y = 1.0$
- Final layer weight updates for learning rate $\mu = 10$
 - $\delta_G = (y - \hat{y})\hat{y}' = (1 - 0.76) \times 0.181 = 0.0434$
 - $\Delta w_{GD} = \mu \delta_G h_D = 10 \times 0.0434 \times 0.90 = 0.391$, new $w_{GD} = 4.891$
 - $\Delta w_{GE} = \mu \delta_G h_E = 10 \times 0.0434 \times 0.17 = 0.074$, new $w_{GE} = -5.126$
 - $\Delta w_{GF} = \mu \delta_G h_F = 10 \times 0.0434 \times 1 = 0.434$, new $w_{GF} = -1.566$



- Hidden node D

- $\delta_D = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y'_D = w_{GD} \delta_G y'_D = 4.5 \times 0.0434 \times 0.0898 = 0.0175$
- $\Delta w_{DA} = \mu \delta_D h_A = 10 \times 0.0175 \times 1.0 = 0.175$
- $\Delta w_{DB} = \mu \delta_D h_B = 10 \times 0.0175 \times 0.0 = 0$
- $\Delta w_{DC} = \mu \delta_D h_C = 10 \times 0.0175 \times 1 = 0.175$



- Hidden node **E**
 - $\delta_E = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y'_E = w_{GE} \delta_G y'_E = -5.2 \times 0.0434 \times 0.2055 = -0.0464$
 - $\Delta w_{EA} = \mu \delta_E h_A = 10 \times -0.0464 \times 1.0 = -0.464$
 - etc.
- this continues until some error criteria are met

Some additional aspects

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

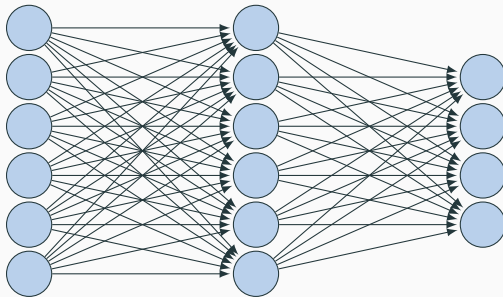
$$\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}} \right]$$

where n is the size of the previous layer

- for deep neural networks

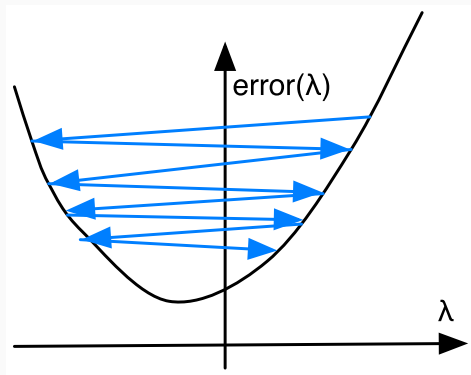
$$\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}} \right]$$

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



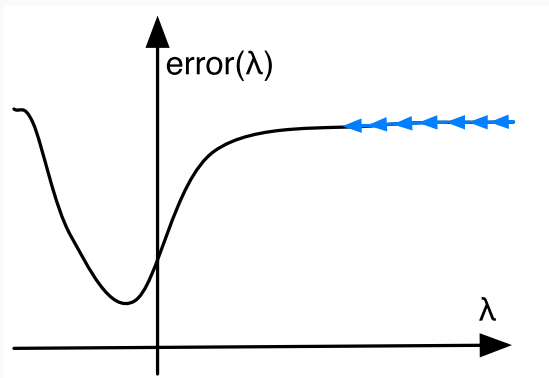
- Predict class: one output node per class
- Training data output: “One-hot vector”, e.g. $\mathbf{y} = (0, 0, 1)^T$
- Prediction
 - predicted class is the output node y_i with highest value

- obtain posterior probability distribution by soft-max, $\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$



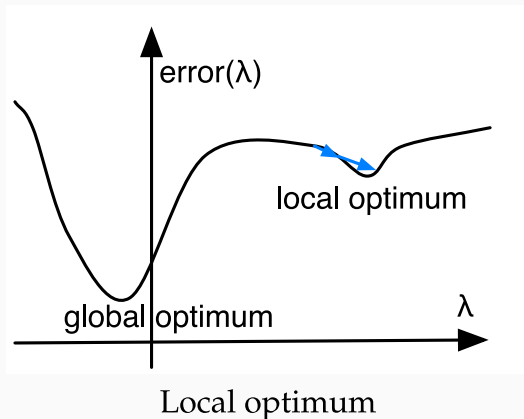
Too high learning rate

Too high learning rate



Bad initialization

Bad initialization



Local optimum

- 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)

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

- 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 $t = g_t = \frac{dE}{dw}$

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

- 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? → bagging, ensemble, ...

- 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 for speed reasons

Computational aspects:

- Forward computation: $\mathbf{s} = \mathbf{W}\mathbf{h}$
- Activation function: $\mathbf{y} = \sigma(\mathbf{s})$
- Error term: $\delta = (\mathbf{y} - \hat{\mathbf{y}}) \cdot \sigma(\mathbf{s})'$
- Propagation of error term: $\delta_i = \mathbf{W}\delta_{i+1} \cdot \sigma(\mathbf{s})'$
- Weight updates: $\Delta\mathbf{W} = \mu\delta\mathbf{h}^\top$

- 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

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

MATLAB: Deep Learning Toolbox

Assignment 5

Use your knowledge of harmonic signal generation and windowing in Matlab to write a script that generates a `.wav` file playing eight tones of the selected music scale.

How do I generate tone frequencies?

- Every octave has 12 half-tones, 8 main tones
- Every octave doubles the frequency, i.e. $f(C_5) = 2f(C_4)$
- Half-tone frequencies form a **geometric series**

tone	C_4	D_4	E_4	F_4	G_4	A_4	B_4	C_5
$f[\text{Hz}]$	261.63	293.66	329.63	349.23	392.00	440.00	493.88	523.25

```
% Define note indexes for C major and G major
```

```
c4=40; cis4=41; d4=42; dis4=43; e4=44; f4=45; fis4=46; g4=47; gis4=48;  
a4=49; ais4=50; h4=51; c5=52; cis5=53; d5=54; dis5=55; e5=56; f5=57;  
fis5=58; g5=59;
```

```
% Octaves
```

```
c_maj = [ c4 d4 e4 f4 g4 a4 h4 c5 ];  
g_maj = [ g4 a4 h4 c5 d5 e5 fis5 g5 ];
```

Example (Frequency of C₄ and C₅ based on A₄)

The tone C₄ is **nine half-tones below** A₄, and the tone C₅ is **three half-tones above** A₄, therefore

$$f(C_4) = f(A_4) \cdot 2^{(-9/12)} = 440 \cdot 2^{-0.75} = 261.63 \text{ Hz}$$

$$f(C_5) = f(A_4) \cdot 2^{(3/12)} = 440 \cdot 2^{0.25} = 523.25 \text{ Hz}$$

Using the `c_maj` defined above, generate one octave of C-major scale:

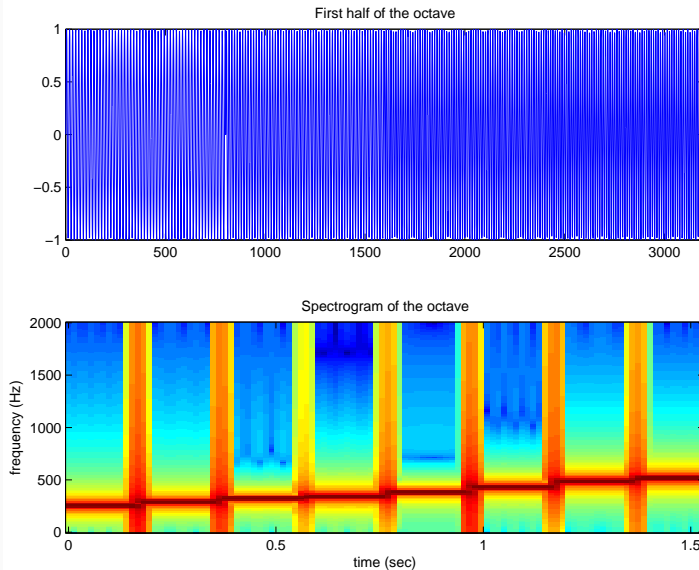
```
Fs = 8000; % sampling frequency
t0 = 0.2; % tone duration
tt = 0:(1/Fs):t0; % sample times
yc = []; % output
for key = c_maj % go through all keys in c_maj
    % compute the tone frequency, freq(a4)=440 Hz
    freq = 440 * (2 .^((key - a4)/12));
    % concatenate the new tone to the previous ones
    yc = [ yc, sin(2*pi*freq*tt) ]; % or cos(2*pi*freq*tt - phi)
end
sound(yc, Fs); % play it with the given sampling frequency
```

The tone transitions are quite disturbing, let's check with spectrogram:

```
Fmax = Fs/4; % maximum frequency
Nfft = 256; % number of FFT coefficients
Nover = 200; % number of overlapping samples
% Display the spectrogram
figure(1);
spectrogram(y, Nfft, Fmax, [], Nover);
title('Spectrogram of the C-major without windowing');
```

The distortion (“clicking”) occurring at tone transition point is due to an abrupt change of the waveform amplitude. This can be mitigated e.g. by applying a window of suitable length to the tone sample.

Assignment 5 — C-major spectrogram with transients



- a) Repeat the steps outlined above for G-major scale, with samples stored in `yg` vector.
- b) Save your sound using `audiowrite('g_major_orig', yg, Fs)`.
- c) It sounds rather artificially, can you find the way to improve the generated record?
- d) **Bonus:** Repeat the steps (a) and (b) for windowed tone samples, write the output to `g_major_window.wav`.
- e) Deliver the code that generates the `.wav` file(s) as a ZIP file by December 23, 2021. No report is necessary.

- a) Repeat the steps outlined above for G-major scale, with samples stored in `yg` vector.
- b) Save your sound using `audiowrite('g_major_orig', yg, Fs)`.
- c) It sounds rather artificially, can you find the way to improve the generated record?
Hint: Use a window function to overlap the tones!
- d) **Bonus:** Repeat the steps (a) and (b) for windowed tone samples, write the output to `g_major_window.wav`.
- e) Deliver the code that generates the `.wav` file(s) as a ZIP file by December 23, 2021. No report is necessary.

- a) Repeat the steps outlined above for G-major scale, with samples stored in `yg` vector.
- b) Save your sound using `audiowrite('g_major_orig', yg, Fs)`.
- c) It sounds rather artificially, can you find the way to improve the generated record?
Hint: Use a window function to overlap the tones! **Hint:** Add harmonics!
- d) **Bonus:** Repeat the steps (a) and (b) for windowed tone samples, write the output to `g_major_window.wav`.
- e) Deliver the code that generates the `.wav` file(s) as a ZIP file by December 23, 2021. No report is necessary.

- a) Repeat the steps outlined above for G-major scale, with samples stored in `yg` vector.
- b) Save your sound using `audiowrite('g_major_orig', yg, Fs)`.
- c) It sounds rather artificially, can you find the way to improve the generated record?
Hint: Use a window function to overlap the tones! **Hint:** Add harmonics! If you want to be even cooler, look at **additive**, **subtractive** or **FM** synthesis.
- d) **Bonus:** Repeat the steps (a) and (b) for windowed tone samples, write the output to `g_major_window.wav`.
- e) Deliver the code that generates the `.wav` file(s) as a ZIP file by December 23, 2021. No report is necessary.

- a) Repeat the steps outlined above for G-major scale, with samples stored in `yg` vector.
- b) Save your sound using `audiowrite('g_major_orig', yg, Fs)`.
- c) It sounds rather artificially, can you find the way to improve the generated record?
Hint: Use a window function to overlap the tones! **Hint:** Add harmonics! If you want to be even cooler, look at **additive**, **subtractive** or **FM** synthesis.
- d) **Bonus:** Repeat the steps (a) and (b) for windowed tone samples, write the output to `g_major_window.wav`.
- e) Deliver the code that generates the `.wav` file(s) as a ZIP file by December 23, 2021. No report is necessary.

Assignment 6

- a) Extend the approach from Assignment 5 to take into account the length of the tones — probably the best solution would be to use an array of tone lengths:

```
jednadve_t = [a4 a4 f4 a4 a4 f4 a4 a4 h4 a4 a4 g4 ]; % tones  
jednadve_d = [ 8 8 4 8 8 4 8 8 8 8 4 4 ]; % tone lengths 1/8 and 1/4 of t0
```

and modify the generator of sample times **tt** to end at **t0/tdiv**:

```
for i = 1:length(jednadve_t) % go through all keys indexes  
    key = jednadve_t(i); % tone number  
    tdiv = jednadve_d(i); % length divisor for the tone  
    tt = 0:(1/Fs):(t0/tdiv); % t0 is now the bar length  
    freq = 440*(2.^((key-a4)/12));  
    sample = sin(2*pi*tt); % can be modified by windowing  
    y = ...  
end
```

- b) Select a Christmas Carol *that has not been composed in C-major scale*.
- c) Compose the Carol using Matlab commands.
Generate the sampled tones using the non-windowed approach from Assignment 5 taking into account different tone lengths as suggested in (a) of this list. Save the result to a **.wav** file.
- d) **Bonus:** Compose a Christmas Carol with a tone scale improved by windowing, harmonics or some other approach mentioned in the bonus part of Assignment 5.
- e) Deliver the code that generates the **.wav** file as a ZIP file by December 23, 2021. No report is necessary.