Introduction to Neural Networks

Mathematical Tools for ITS (11MAI)

Mathematical tools, 2021

Jan Přikryl
(based on the book "Introduction to Statistical Learning", `https://www.statlearning.com/`)

11MAI, lecture 9

Monday, November 15, 2021

version: 2021-11-15 13:15

Department of Applied Mathematics, CTU FTS

Artificial Neural Networks

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.

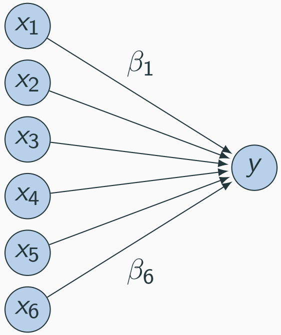
Part of success due to vast improvements in computing power (GPUs), larger training sets, and free software (e.g., Tensorflow and PyTorch, plus Keras).

Linear Models

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”



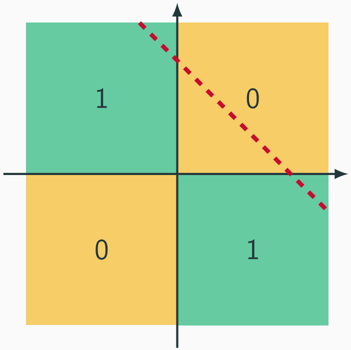
Limits of Linearity

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).

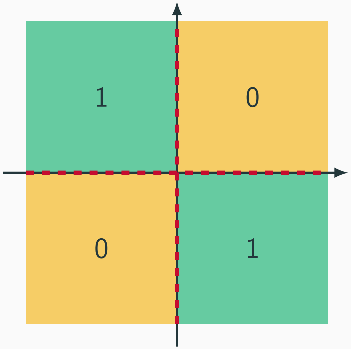
XOR

Linear models cannot model XOR-like behaviour:



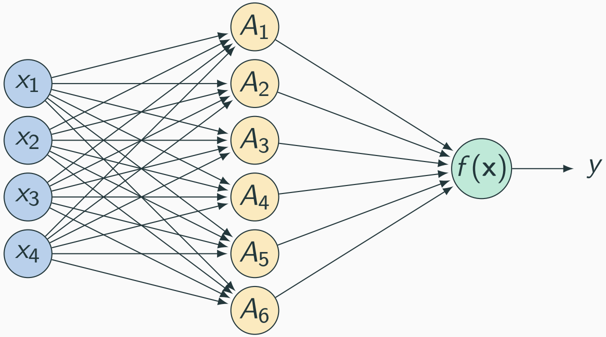
XOR

Linear models cannot model XOR-like behaviour:



Network with Multiple Layers

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



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

Network details

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, 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

Adding non-Linearity

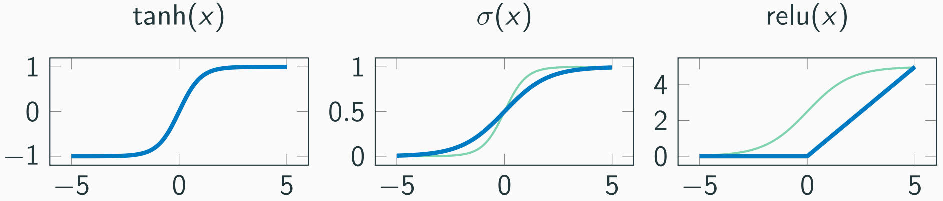
Instead of computing a linear combination

$$\text{score}(\boldsymbol{\beta}, \mathbf{d}) = \beta 0 + \sum_j \beta_j h_j(\mathbf{d})$$

use a non-linear activation function

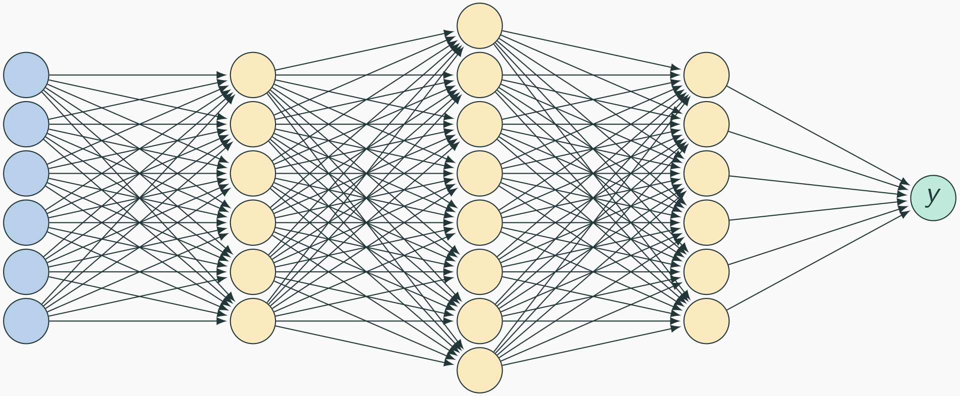
$$\text{score}(\mathbf{w}, \mathbf{d}) = g\left(w_0 + \sum_j w_j h_j(\mathbf{d})\right)$$

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



Deep Learning

More layers = deep learning



What Depths Holds

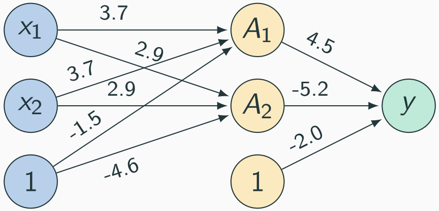
We can interpret the deep NN as follows:

- 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.

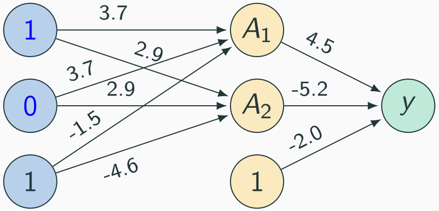
Example

Simple Neural Network



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

Simple Neural Network

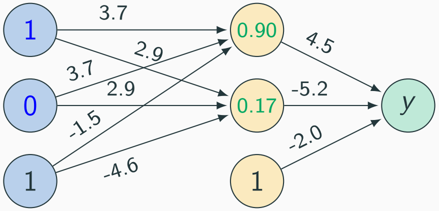


- Try out two input values
- Hidden unit computation

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

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

Simple Neural Network

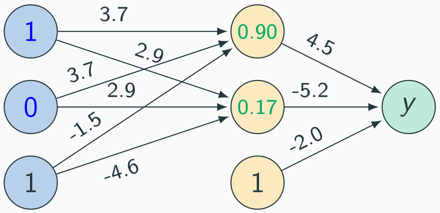


- Try out two input values
- Hidden unit computation

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

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

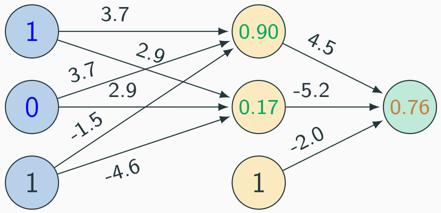
Simple Neural Network



- Output unit computation

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

Simple Neural Network



- 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 for all Binary Inputs

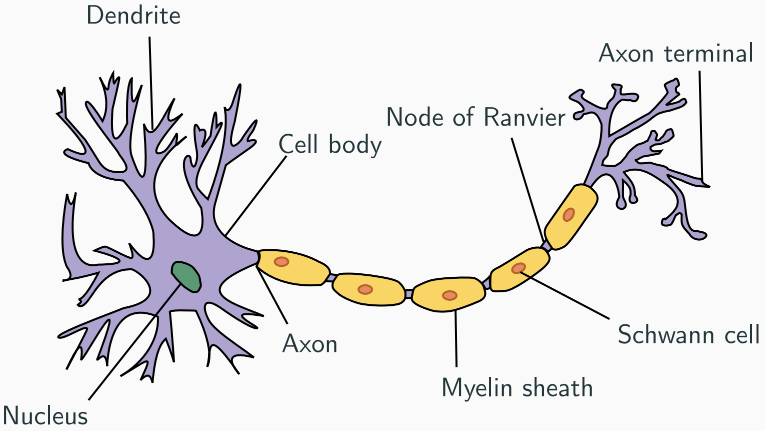
Input x_1	Input x_2	Hidden A_1	Hidden A_2	Output y
0	0	0.12	0.02	$0.18 \rightarrow 0$
0	1	0.88	0.27	$0.74 \rightarrow 1$
1	0	0.73	0.12	$0.74 \rightarrow 1$
1	1	0.99	0.73	$0.33 \rightarrow 0$

- 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?

Neuron in the Brain

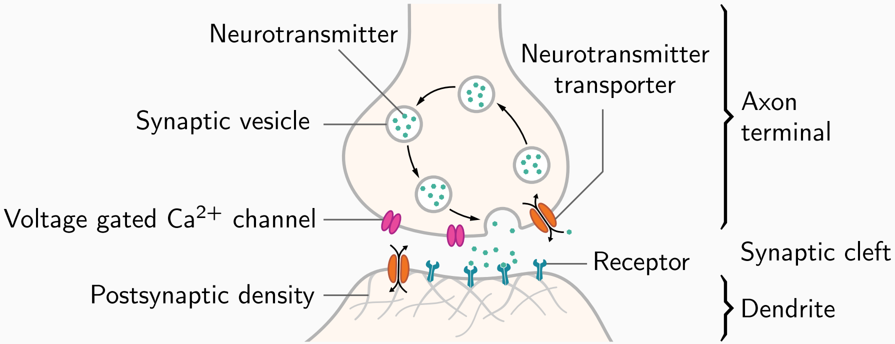
- The human brain is made up of about 100 billion neurons



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

Neural Communication

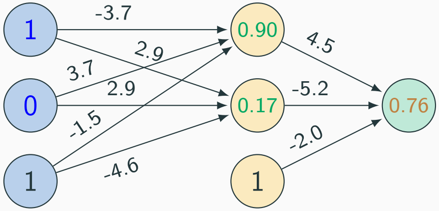
The axon of the neuron is connected to the dendrites of many other neurons



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

Error



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?

Key Concepts

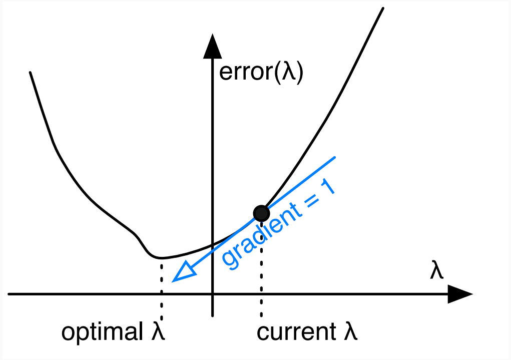
Find minimal training error using non-linear minimization, e.g. by gradient descent:

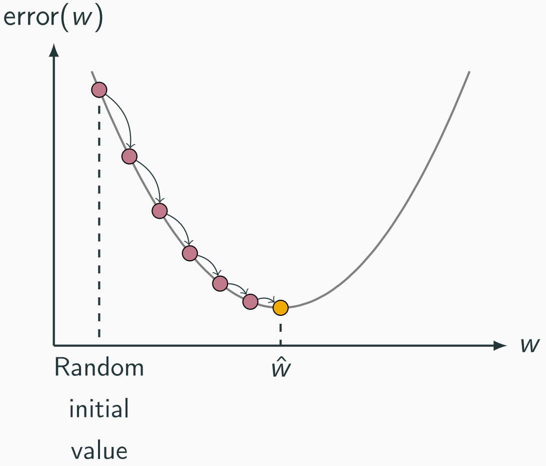
- error is a function of the weights
- we want to reduce the error
- move towards the error minimum
- gradient descent: compute gradient $\rightarrow$ get direction to the error minimum
- requires (numerical) derivative of the error function

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

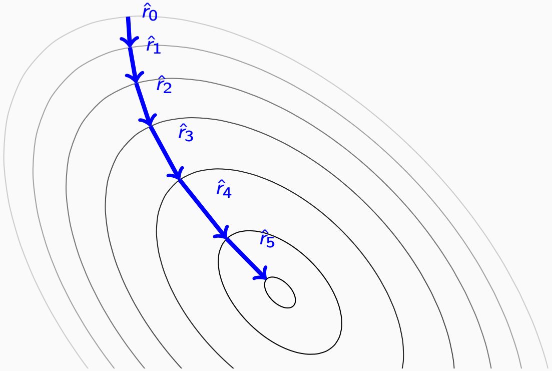
- first adjust last set of weights
- propagate error back to each previous layer
- adjust their weights

Gradient Descent





Gradient Descent



Derivative of Sigmoid

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

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

Final Layer Update

- 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

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

Final Layer Update

- 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

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

- Error E is defined with respect to y

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

Final Layer Update

- 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

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

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

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

Final Layer Update

- 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

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

- 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

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

  - error rate
  - derivative of sigmoid: $\hat{y}'$

- Weight adjustment will be scaled by a fixed learning rate μ:

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

Multiple Output Nodes

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

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

Hidden Layer Update

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

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

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

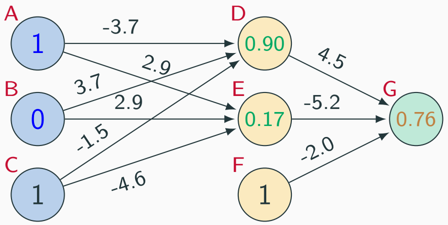
- Back-propagate the error term (why this way? there is math to back it up . . .)

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

- Universal update formula

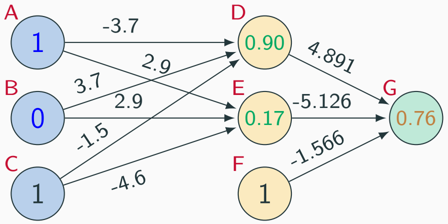
$$\Delta w_{j\leftarrow k} = \mu\delta_j h_k$$

Our Example



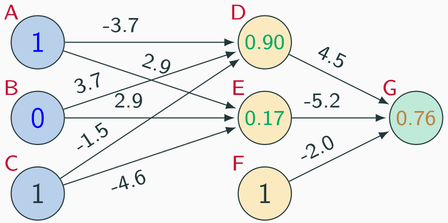
- Computed output: $\hat{y} = 0.76$
- Correct output: $y = 1.0$
- Final layer weight updates for learning rate $\mu = 10$
  - $\delta_\mathrm{G} = (y - \hat{y})\hat{y}' = (1 - 0.76) \times 0.181 = 0.0434$
  - $\Delta w_\mathrm{GD} = \mu \delta_\mathrm{G} h_\mathrm{D} = 10 \times 0.0434 \times 0.90 = 0.391$
  - $\Delta w_\mathrm{GE} = \mu \delta_\mathrm{G} h_\mathrm{E} = 10 \times 0.0434 \times 0.17 = 0.074$
  - $\Delta w_\mathrm{GF} = \mu \delta_\mathrm{G} h_\mathrm{F} = 10 \times 0.0434 \times 1 = 0.434$

Our Example



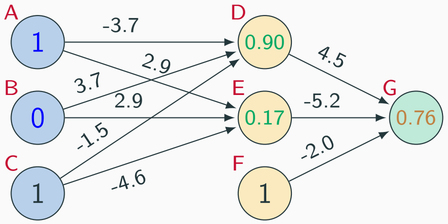
- Computed output: $\hat{y} = 0.76$
- Correct output: $y = 1.0$
- Final layer weight updates for learning rate $\mu = 10$
    - $\delta_\mathrm{G} = (y - \hat{y})\hat{y}' = (1 - 0.76) \times 0.181 = 0.0434$
    - $\Delta w_\mathrm{GD} = \mu \delta_\mathrm{G} h_\mathrm{D} = 10 \times 0.0434 \times 0.90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

Hidden Layer Updates



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

Hidden Layer Updates



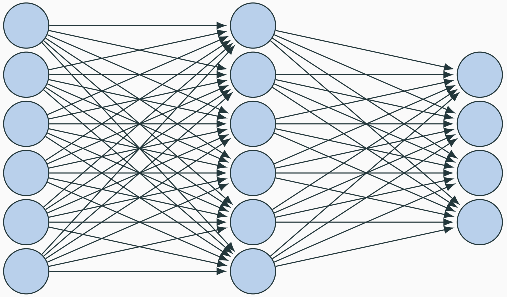
- Hidden node **E**
  - $\delta_{\mathrm{E}} = \left(\sum_j w_{j\leftarrow i}\delta_j\right) y'_{\mathrm{E}} = w_{\mathrm{GE}}\delta_{\mathrm{G}} y'_{\mathrm{E}} = -5.2 \times 0.0434 \times 0.2055 = -0.0464$
  - $\Delta w_{\mathrm{EA}} = \mu \delta_{\mathrm{E}} h_{\mathrm{A}} = 10 \times -0.0464 \times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

Some additional aspects

Initialization of Weights

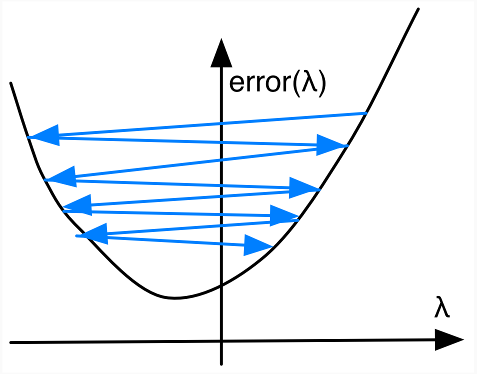
- Weights are initialized randomly, e.g. uniformly from interval $[-0.01, 0.01]$
- Glorot and Bengio (2010) suggest
  - for shallow neural networks
    $$\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$
    where n is the size of the previous layer
  - for deep neural networks
    $$\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$
    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 the output node y_i with highest value
  - obtain posterior probability distribution by soft-max, $\mathrm{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$

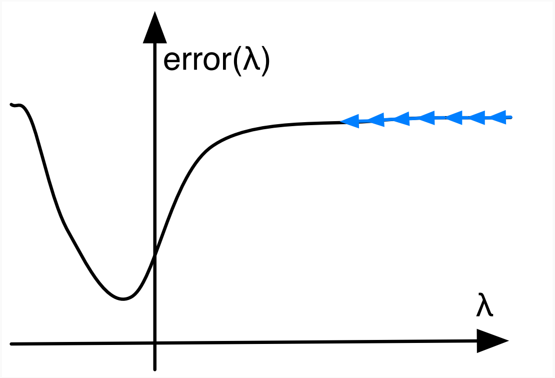
Problems with Gradient Descent Training



Too high learning rate

Too high learning rate

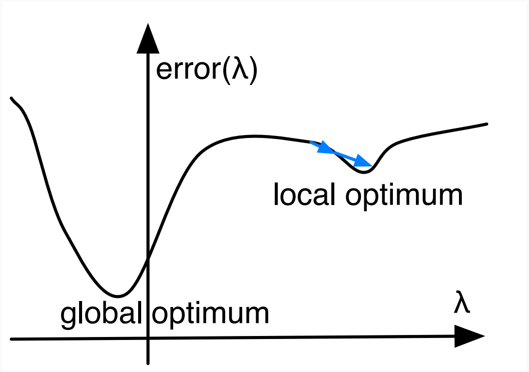
Problems with Gradient Descent Training



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

- 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

Vector and Matrix Multiplications

Computational aspects:

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

GPU

- Neural network layers may have, say, 200 nodes
- Computations such as $\mathbf{s} = \mathbf{W}\mathbf{h}$ 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