Neural Networks

Philipp Koehn

14 April 2020

Supervised Learning

- Examples described by attribute values (Boolean, discrete, continuous, etc.)

- E.g., situations where I will/won't wait for a table:

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

- Classification of examples is positive (T) or negative (F)

Naive Bayes Models

- Bayes rule

$$p(C|\mathbf{A}) = \frac{1}{Z}\, p(\mathbf{A}|C)\, p(C)$$

- Independence assumption

$$\begin{aligned} p(\mathbf{A}|C) &= p(a_1, a_2, a_3, ..., a_n|C) \\ &\simeq \prod_i p(a_i|C) \end{aligned}$$

- Weights

$$p(\mathbf{A}|C) = \prod_i p(a_i|C)^{\lambda_i}$$

Naive Bayes Models

- Linear model

$$p(\mathbf{A}|C) = \prod_i p(a_i|C)^{\lambda_i}$$
$$= \exp \sum_i \lambda_i \log p(a_i|C)$$

- Probability distribution as features

$$h_i(\mathbf{A}, C) = \log p(a_i|C)$$
$$h_0(\mathbf{A}, C) = \log p(C)$$

- Linear model with features

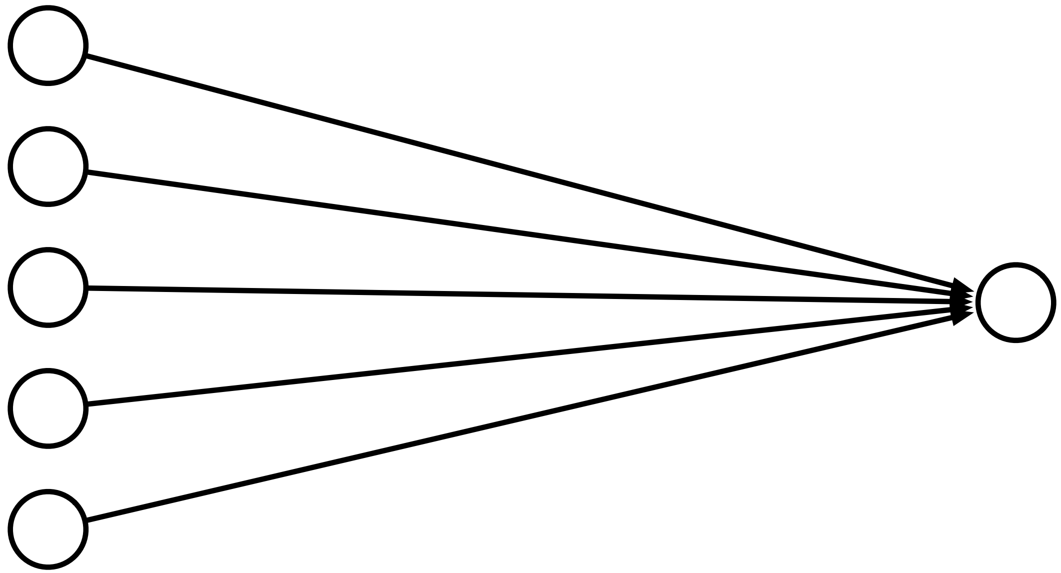
$$p(C|\mathbf{A}) \propto \sum_i \lambda_i \, h_i(\mathbf{A}, C)$$

Linear Model

- Weighted linear combination of feature values h_j and weights λ_j for example $\mathbf{d}_i$

$$\text{score}(\lambda, \mathbf{d}_i) = \sum_j \lambda_j \, h_j(\mathbf{d}_i)$$

- Such models can be illustrated as a "network"



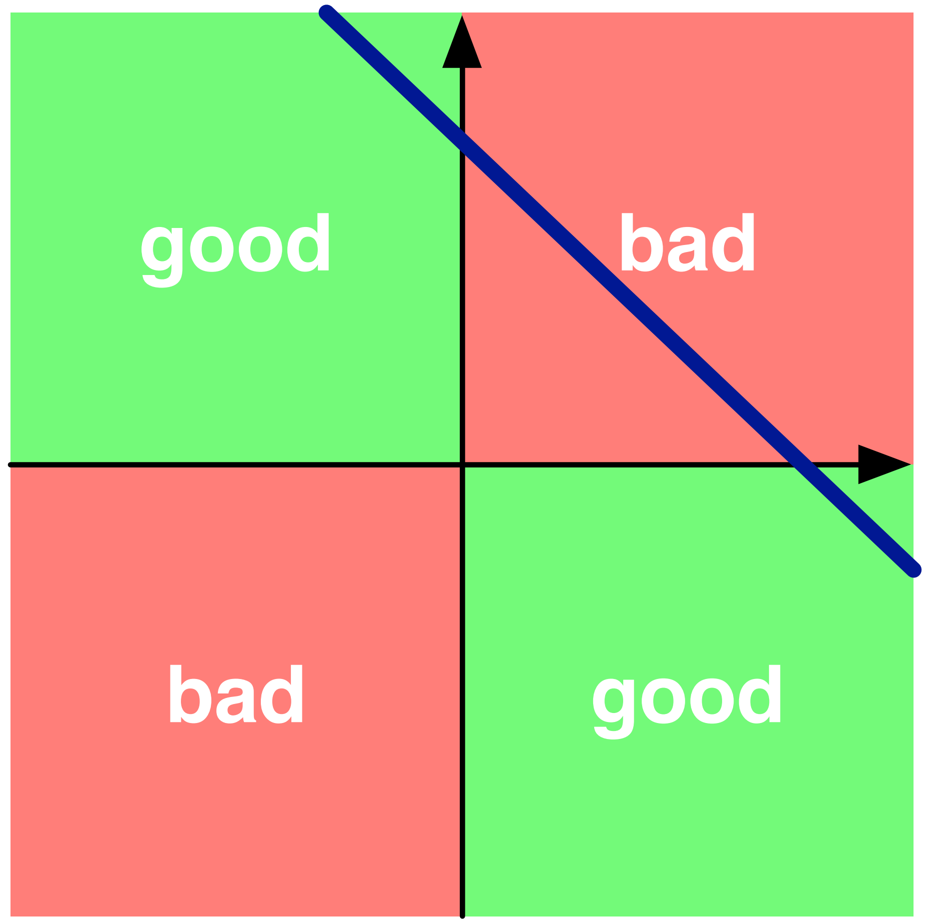
Limits of Linearity

- 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

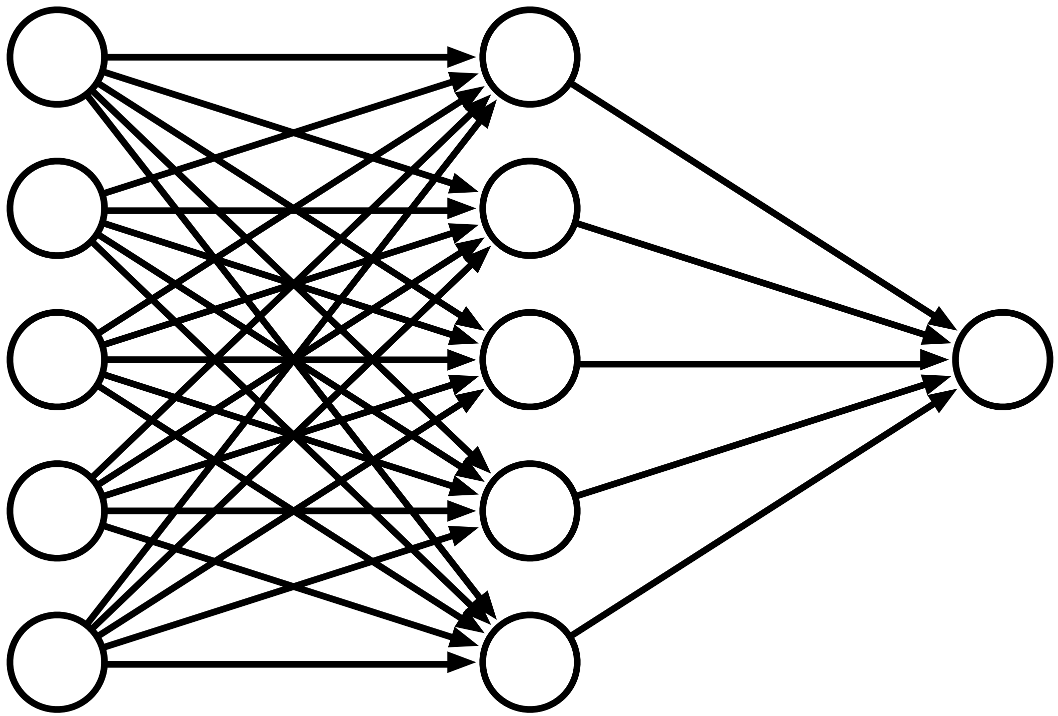
XOR

- Linear models cannot model XOR



Multiple Layers

- 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

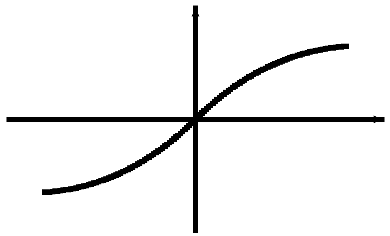
$$\text{score}(\lambda, \mathbf{d}_i) = \sum_j \lambda_j \, h_j(\mathbf{d}_i)$$

- Add a non-linear function

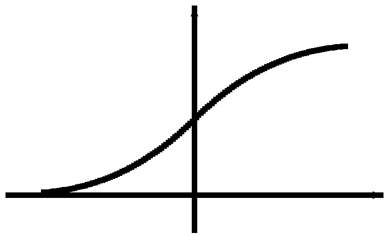
$$\text{score}(\lambda, \mathbf{d}_i) = f\Big(\sum_j \lambda_j \, h_j(\mathbf{d}_i) \Big)$$

- Popular choices

tanh(x)



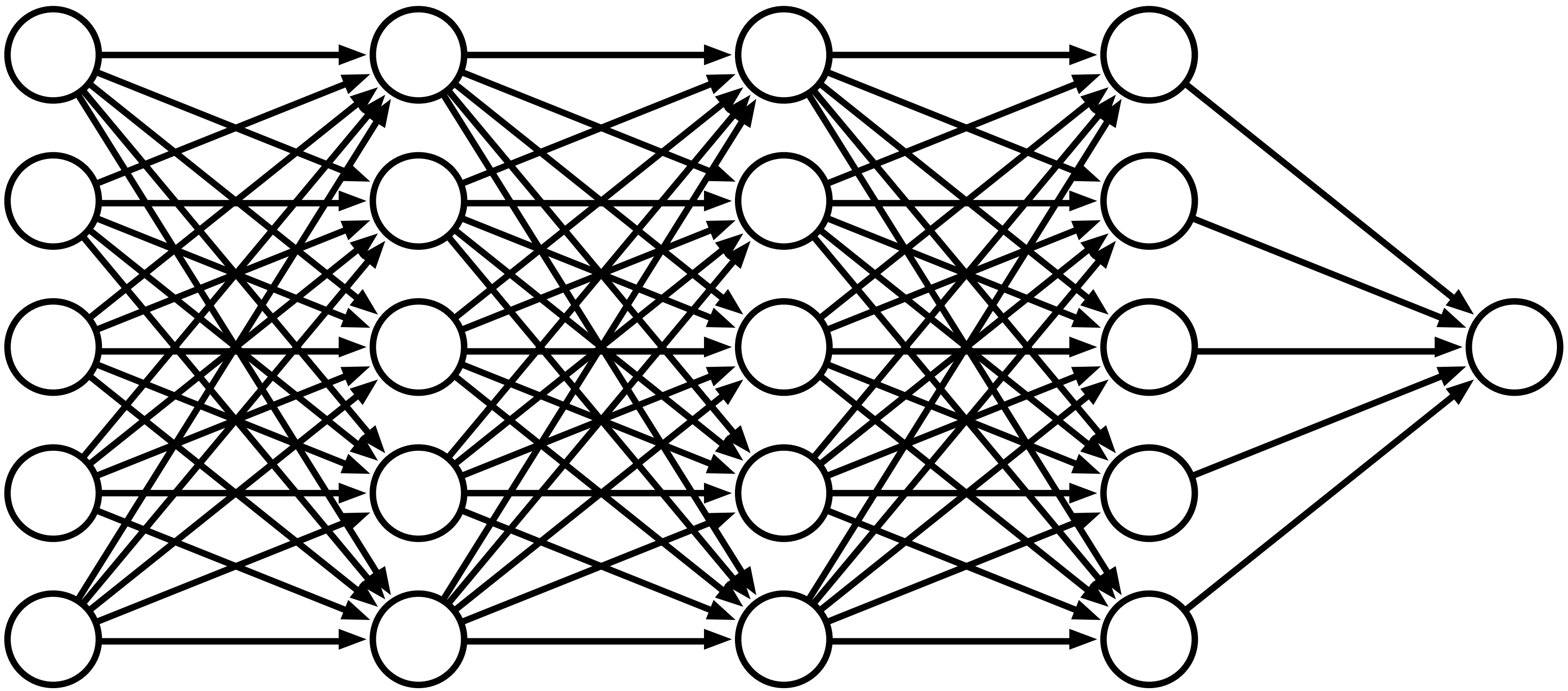
sigmoid(x) = $\frac{1}{1+e^{-x}}$



(sigmoid is also called the "logistic function")

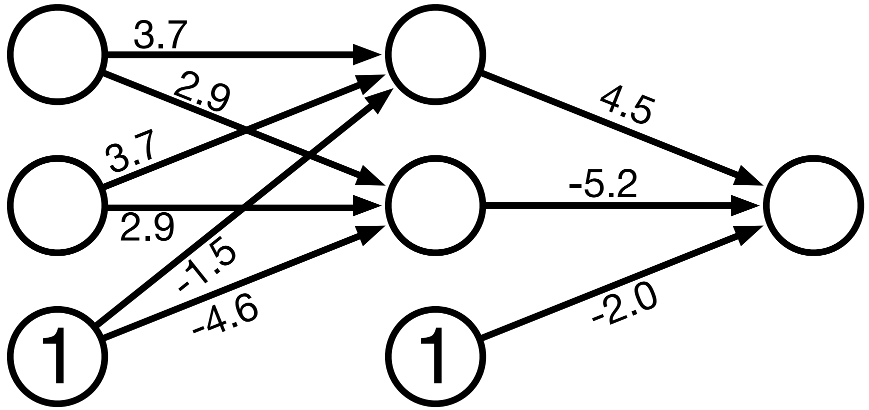
Deep Learning

- More layers = deep learning



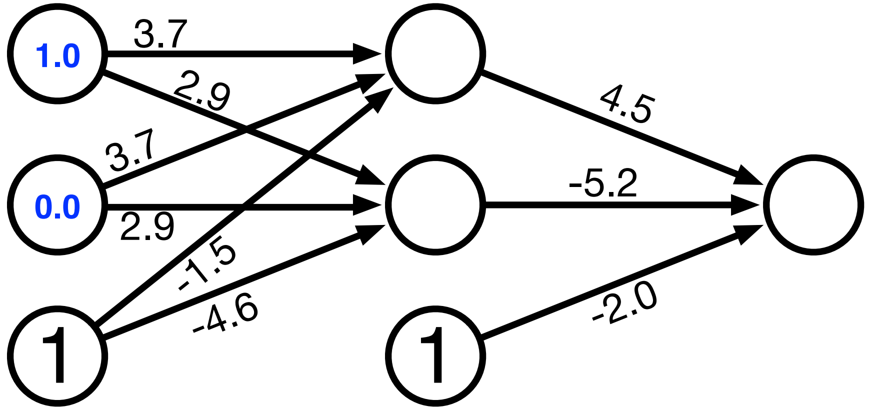
example

Simple Neural Network



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

Sample Input

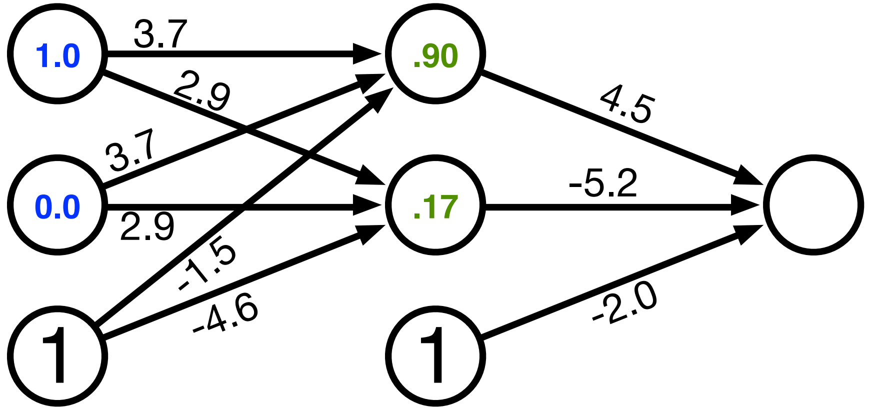


- Try out two input values
- Hidden unit computation

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

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

Computed Hidden

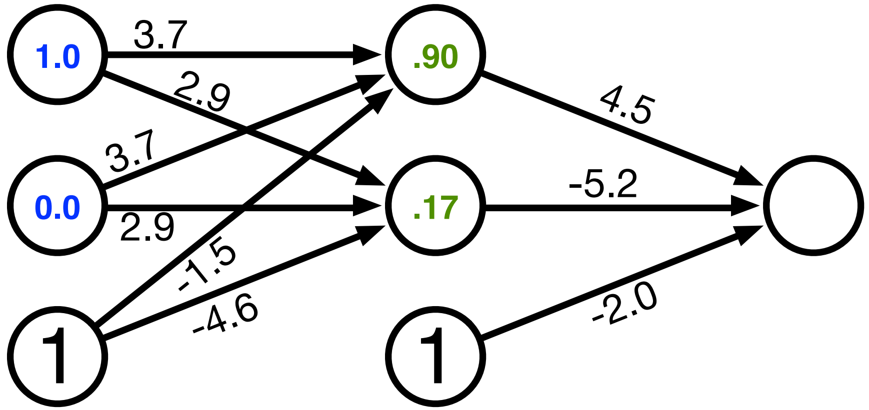


- Try out two input values
- Hidden unit computation

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

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

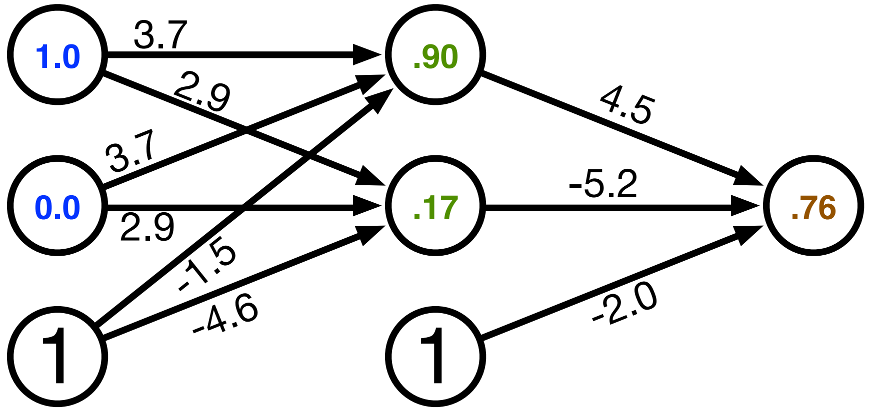
Compute Output



- Output unit computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1+e^{-1.17}} = 0.76$$

Computed Output



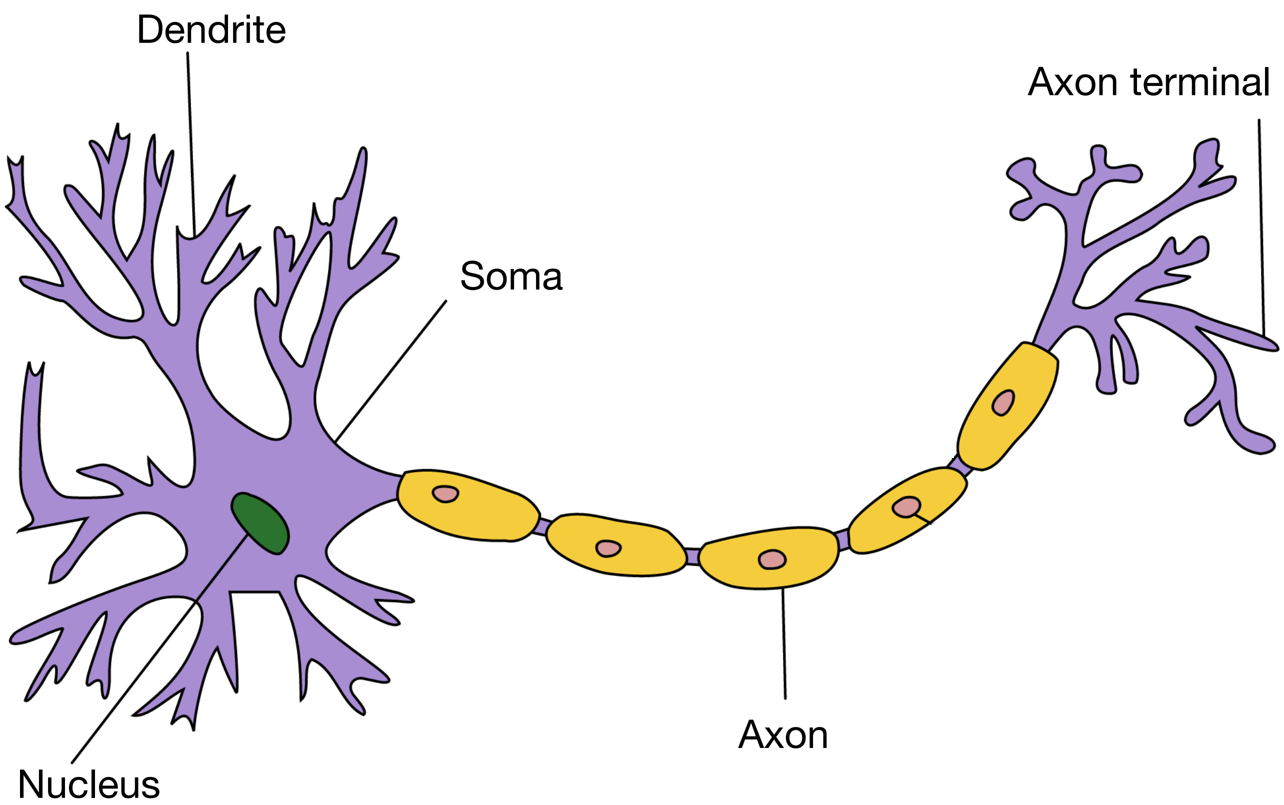
- Output unit computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

why ”neural” networks?

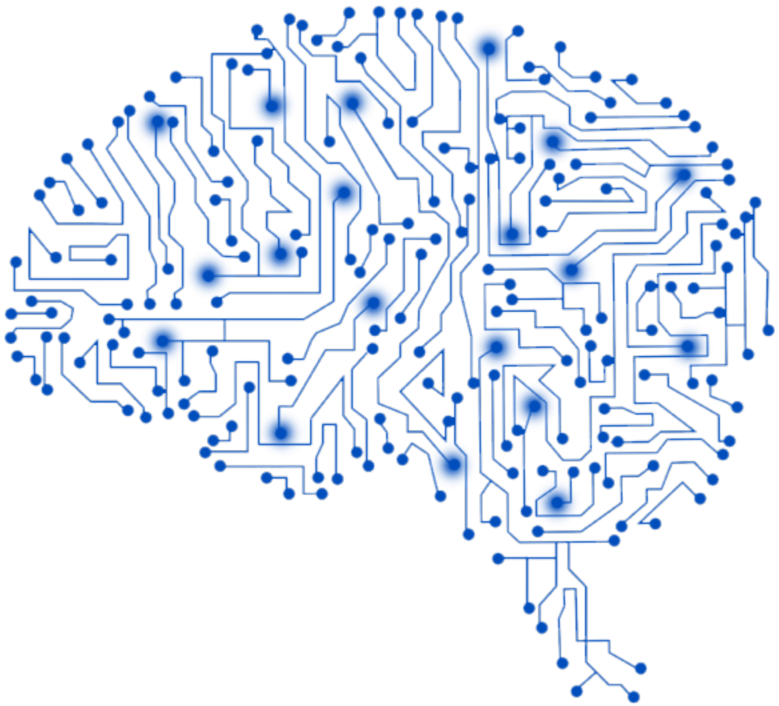
Neuron in the Brain

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



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

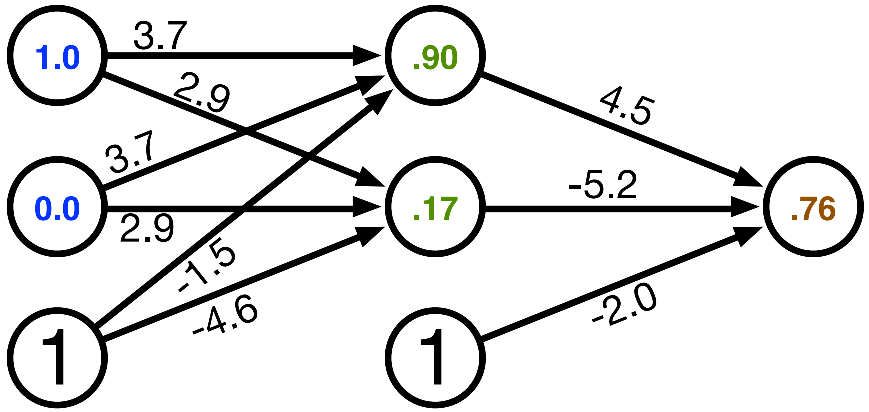
The Brain vs. Artificial Neural Networks



- 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

back-propagation training

Error



- Computed output: $y = .76$
- Correct output: $t = 1.0$

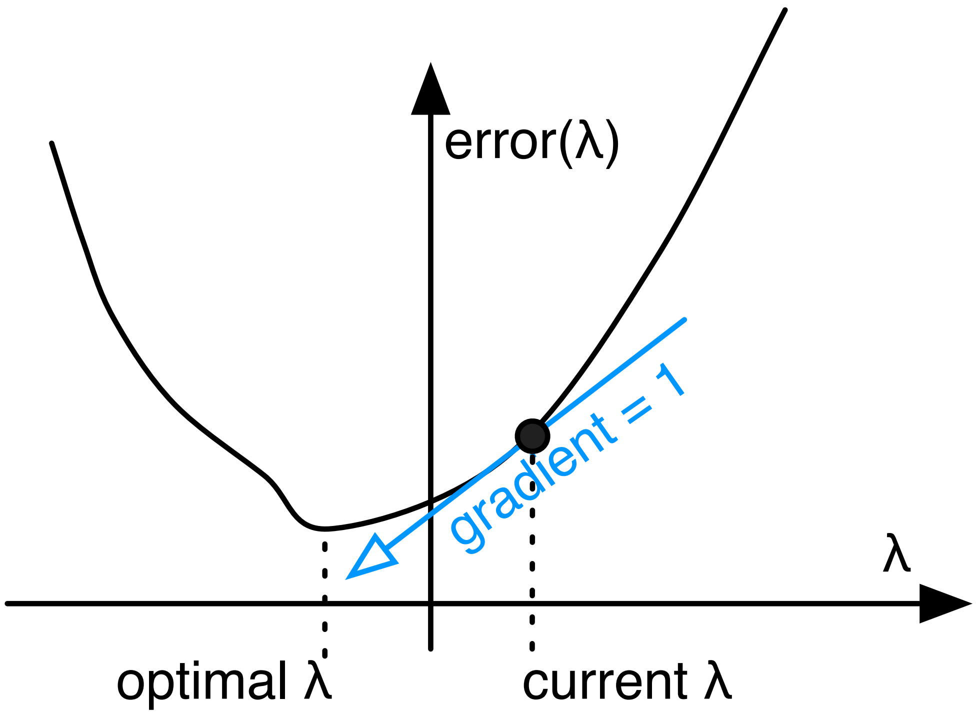
⇒ How do we adjust the weights?

Key Concepts

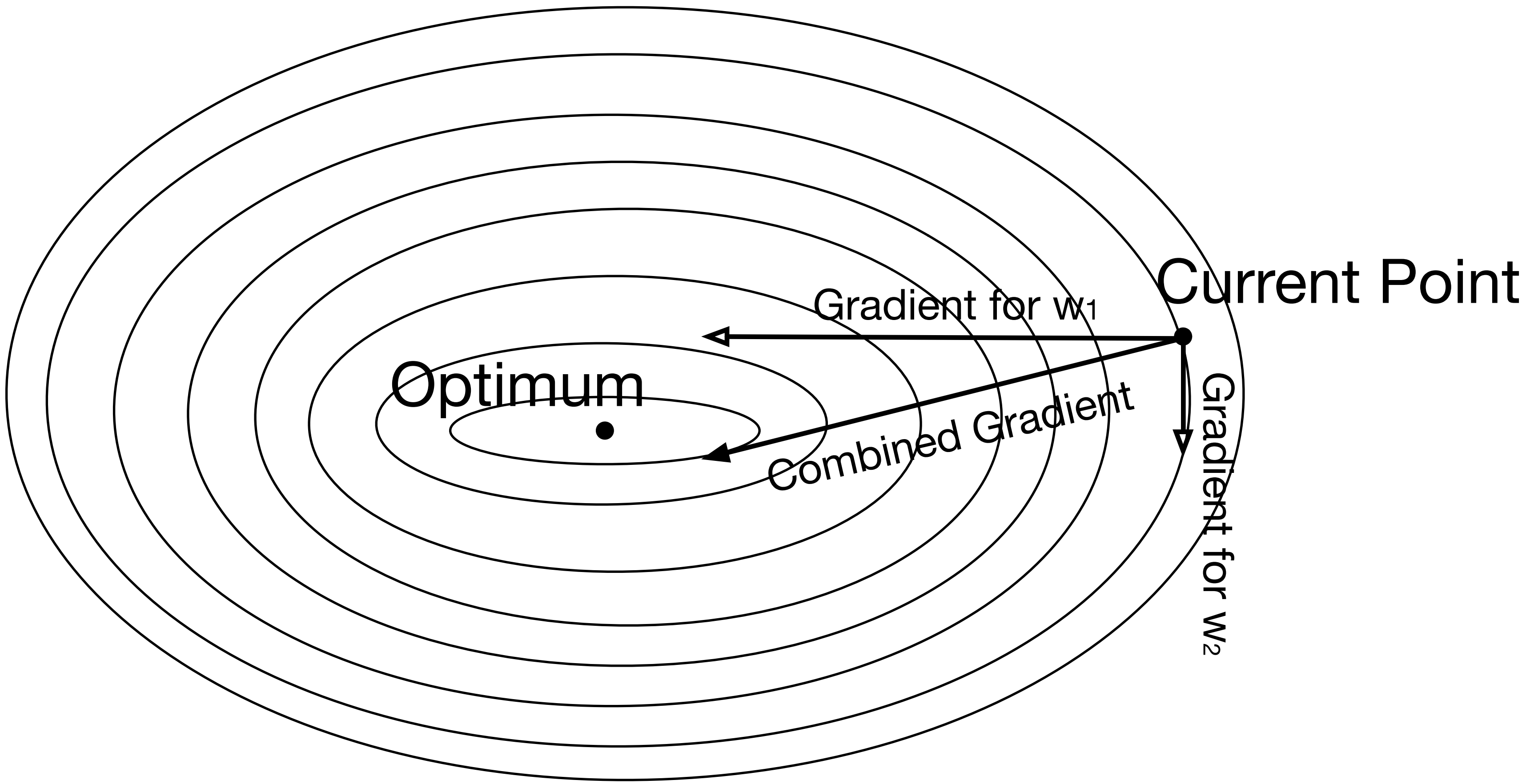
- Gradient descent
  - error is a function of the weights
  - we want to reduce the error
  - gradient descent: move towards the error minimum
  - compute gradient $\rightarrow$ get direction to the error minimum
  - adjust weights towards direction of lower error

- 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

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

- Reminder: quotient rule

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{g(x)f'(x) - f(x)g'(x)}{g(x)^2}$$

- Derivative

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

Final Layer Update

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

Final Layer Update (1)

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

- Error E is defined with respect to y

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

Final Layer Update (2)

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- 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 with respect to x is $\text{sigmoid}(s)$

$$\frac{dy}{ds} = \frac{d\,\text{sigmoid}(s)}{ds} = \text{sigmoid}(s)(1 - \text{sigmoid}(s)) = y(1 - y)$$

Final Layer Update (3)

- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

- x is weighted linear combination of hidden node values h_k

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

Putting it All Together

- Derivative of error with regard to one weight w_k

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

  - 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 $k \rightarrow j$ 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

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

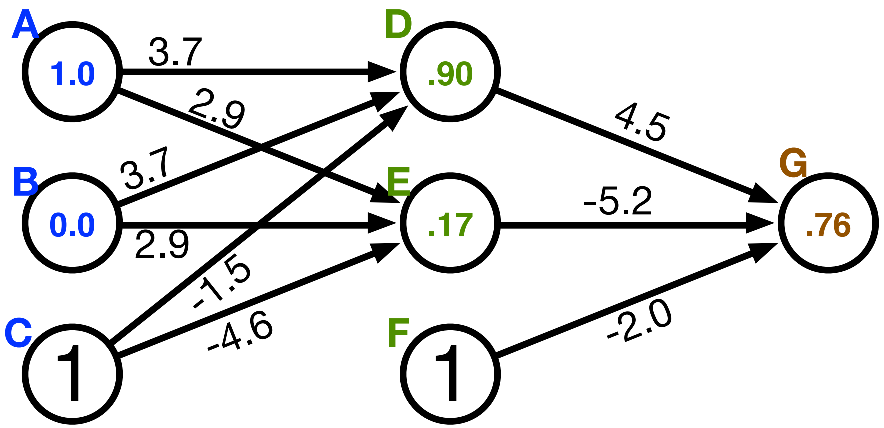
- Back-propagate the error term
  (why this way? there is math to back it up...)

$$\delta_i = \Big(\sum_j w_{j \leftarrow i} \delta_j \Big)\ y'_i$$

- Universal update formula

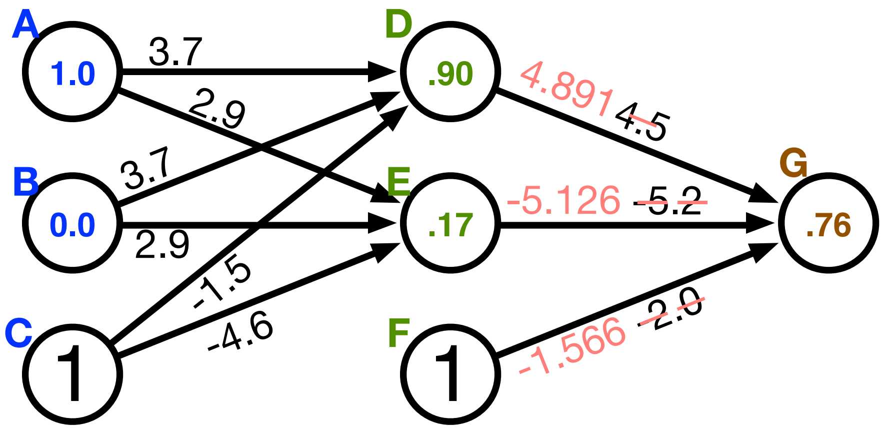
$$\Delta w_{j \leftarrow k} = \mu\ \delta_j\ h_k$$

Our Example



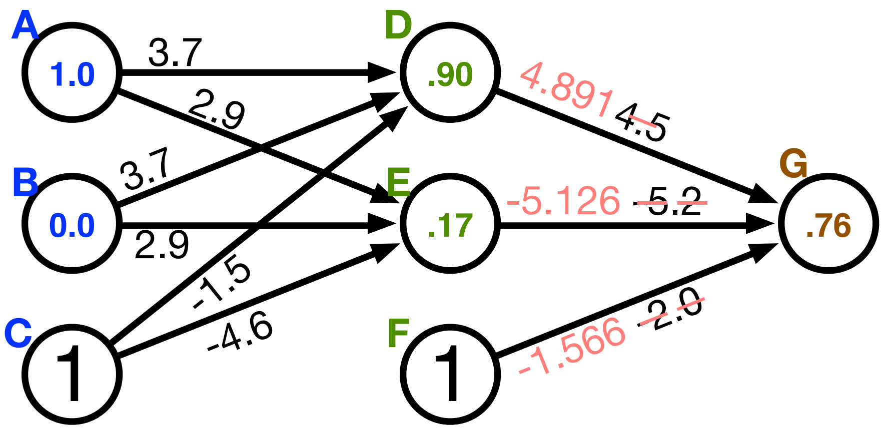
- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_{\text{G}} = (t - y)\ y' = (1 - .76)\ 0.181 = .0434$
  - $\Delta w_{\text{GD}} = \mu\ \delta_{\text{G}}\ h_{\text{D}} = 10 \times .0434 \times .90 = .391$
  - $\Delta w_{\text{GE}} = \mu\ \delta_{\text{G}}\ h_{\text{E}} = 10 \times .0434 \times .17 = .074$
  - $\Delta w_{\text{GF}} = \mu\ \delta_{\text{G}}\ h_{\text{F}} = 10 \times .0434 \times 1 = .434$

Our Example



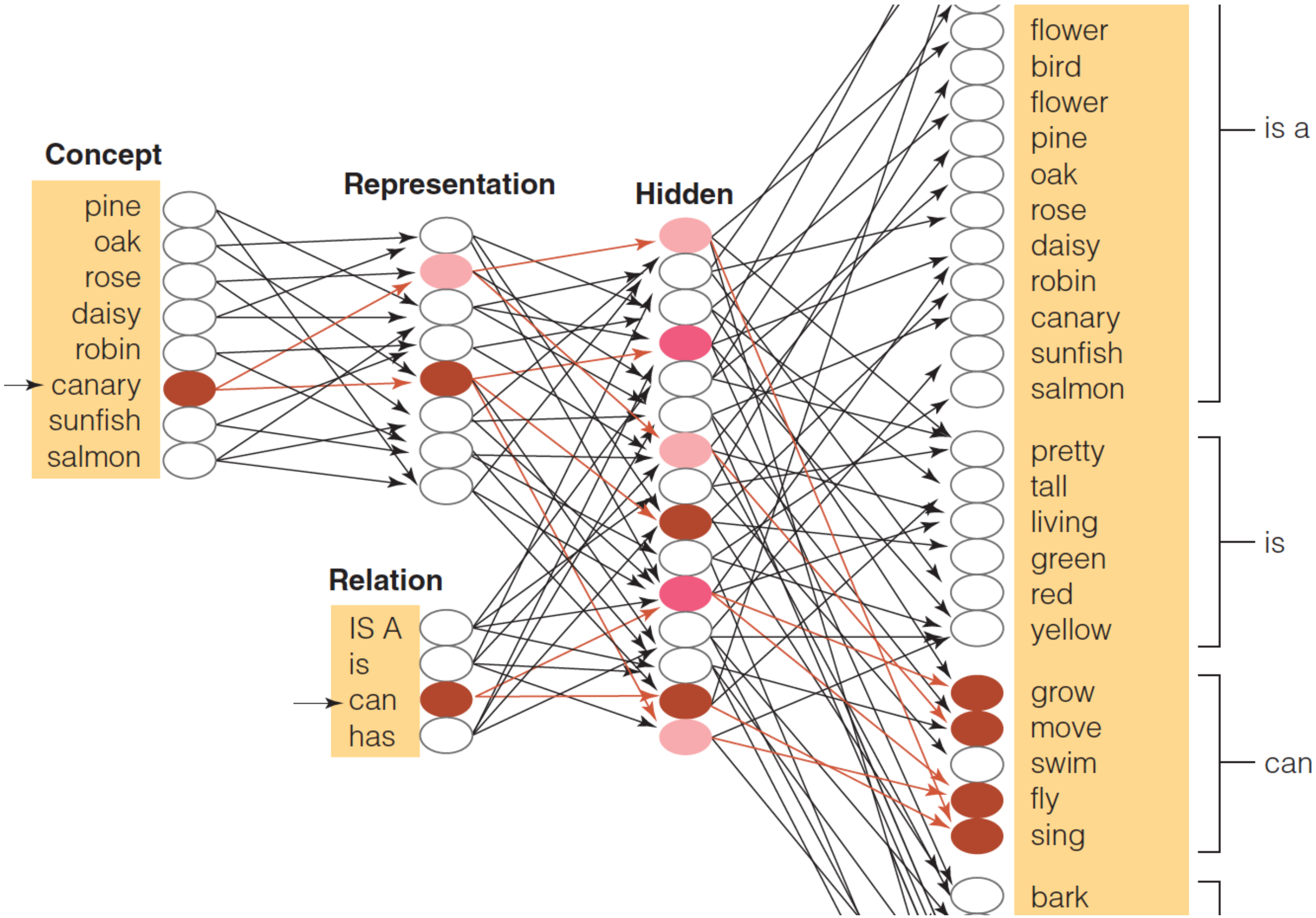
- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_\text{G} = (t - y)\ y' = (1 - .76)\ 0.181 = .0434$
  - $\Delta w_\text{GD} = \mu\ \delta_\text{G}\ h_\text{D} = 10 \times .0434 \times .90 = .391$
  - $\Delta w_\text{GE} = \mu\ \delta_\text{G}\ h_\text{E} = 10 \times .0434 \times .17 = .074$
  - $\Delta w_\text{GF} = \mu\ \delta_\text{G}\ h_\text{F} = 10 \times .0434 \times 1 = .434$

Hidden Layer Updates



- Hidden node **D**
  - $\delta_{\text{D}} = \left(\sum_j w_{j\leftarrow i}\delta_j \right) y'_{\text{D}} = w_{\text{GD}}\ \delta_{\text{G}}\ y'_{\text{D}} = 4.5 \times .0434 \times .0898 = .0175$
  - $\Delta w_{\text{DA}} = \mu\ \delta_{\text{D}}\ h_{\text{A}} = 10 \times .0175 \times 1.0 = .175$
  - $\Delta w_{\text{DB}} = \mu\ \delta_{\text{D}}\ h_{\text{B}} = 10 \times .0175 \times 0.0 = 0$
  - $\Delta w_{\text{DC}} = \mu\ \delta_{\text{D}}\ h_{\text{C}} = 10 \times .0175 \times 1 = .175$
- Hidden node **E**
  - $\delta_{\text{E}} = \left(\sum_j w_{j\leftarrow i}\delta_j \right) y'_{\text{E}} = w_{\text{GE}}\ \delta_{\text{G}}\ y'_{\text{E}} = -5.2 \times .0434 \times 0.1411 = -.0318$
  - $\Delta w_{\text{EA}} = \mu\ \delta_{\text{E}}\ h_{\text{A}} = 10 \times -.0318 \times 1.0 = -.318$
  - etc.

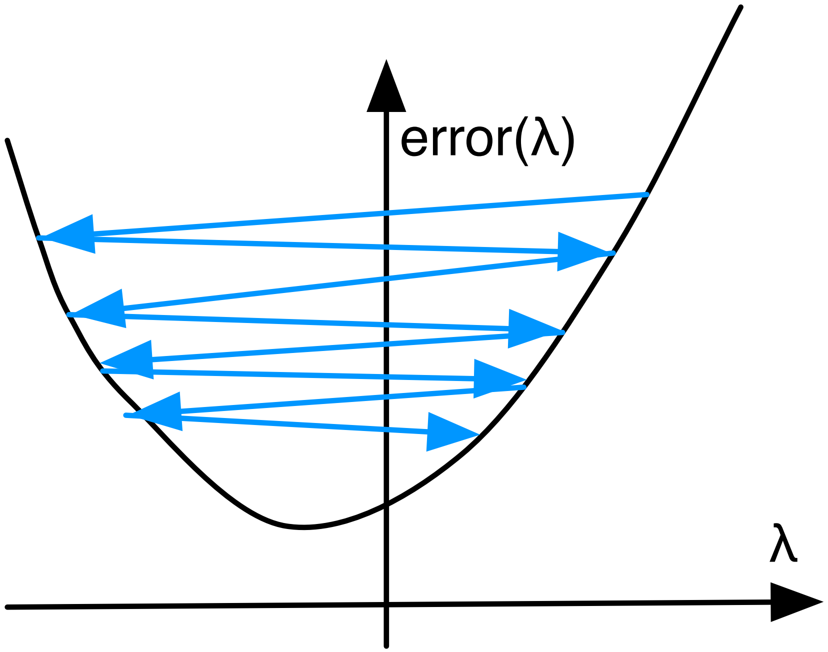
Connectionist Semantic Cognition



- Hidden layer representations for concepts and concept relationships

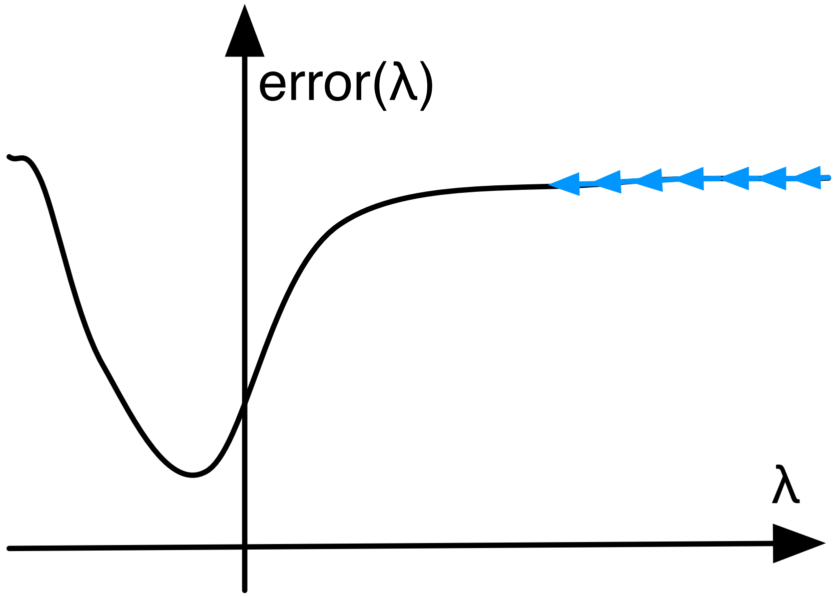
some additional aspects

Problems with Gradient Descent Training



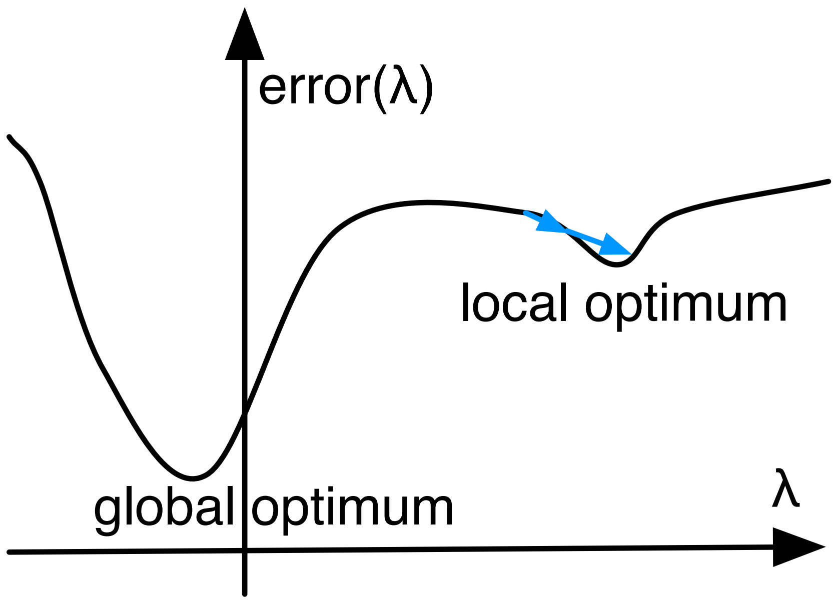
Too high learning rate

Problems with Gradient Descent Training



Bad initialization

Problems with Gradient Descent Training



Local optimum

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

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

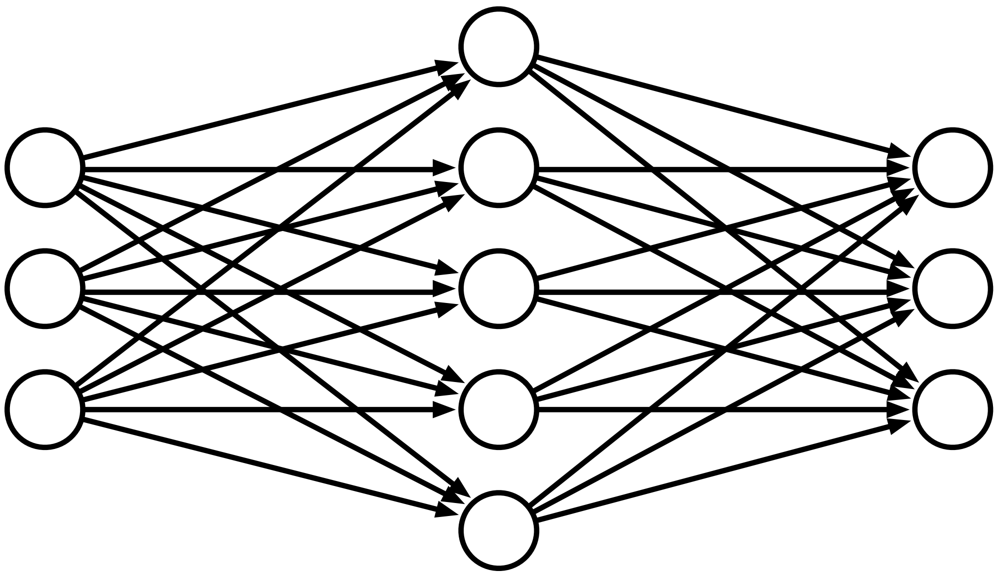
    n is the size of the previous layer

  - for deep neural networks

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

    n_j is the size of the previous layer, n_j size of next layer

Neural Networks for Classification



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

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

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)

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

computational aspects

Vector and Matrix Multiplications

- Forward computation: $\vec{s} = W\vec{h}$

- Activation function: $\vec{y} = \text{sigmoid}(\vec{h})$

- Error term: $\vec{\delta} = (\vec{t} - \vec{y}) \text{ sigmoid}'(\vec{s})$

- Propagation of error term: $\vec{\delta}_i = W\vec{\delta}_{i+1} \cdot \text{sigmoid}'(\vec{s})$

- Weight updates: $\Delta W = \mu\vec{\delta}\vec{h}^T$

GPU

- Neural network layers may have, say, 200 nodes
- Computations such as $W\vec{h}$ require $200 \times 200 = 40,000$ multiplications
- Graphics Processing Units (GPU) are designed for such computations
  - image rendering requires such vector and matrix operations
  - massively mulit-core but lean processing units
  - example: NVIDIA Tesla K20c GPU provides 2496 thread processors
- Extensions to C to support programming of GPUs, such as CUDA

Toolkits

- Tensorflow (Google)
- PyTorch (Facebook)
- MXNet (Amazon)