Universität Stuttgart
KI – Institute for Artificial Intelligence

Analytic Computing

Machine Learning
11 Neural Networks Part 1

Prof. Dr. Steffen Staab

Nadeen Fatallah
Daniel Frank
Akram Sadat Hosseini
Jiaxin Pan
Osama Mohamed
Arvindh Arunbabu
Tim Schneider
Yi Wang

https://www.ki.uni-stuttgart.de/

Learning Objectives

- Biological vs artificial neural networks
- Feed-forward networks
- XOR-Problem
- Activation functions
- Loss functions for binomial, multi-nomial distributions
- Computation graphs
- Backpropagation

1 Biological vs artificial neural networks

History of Machine Intelligence (Artificial Neural Networks)

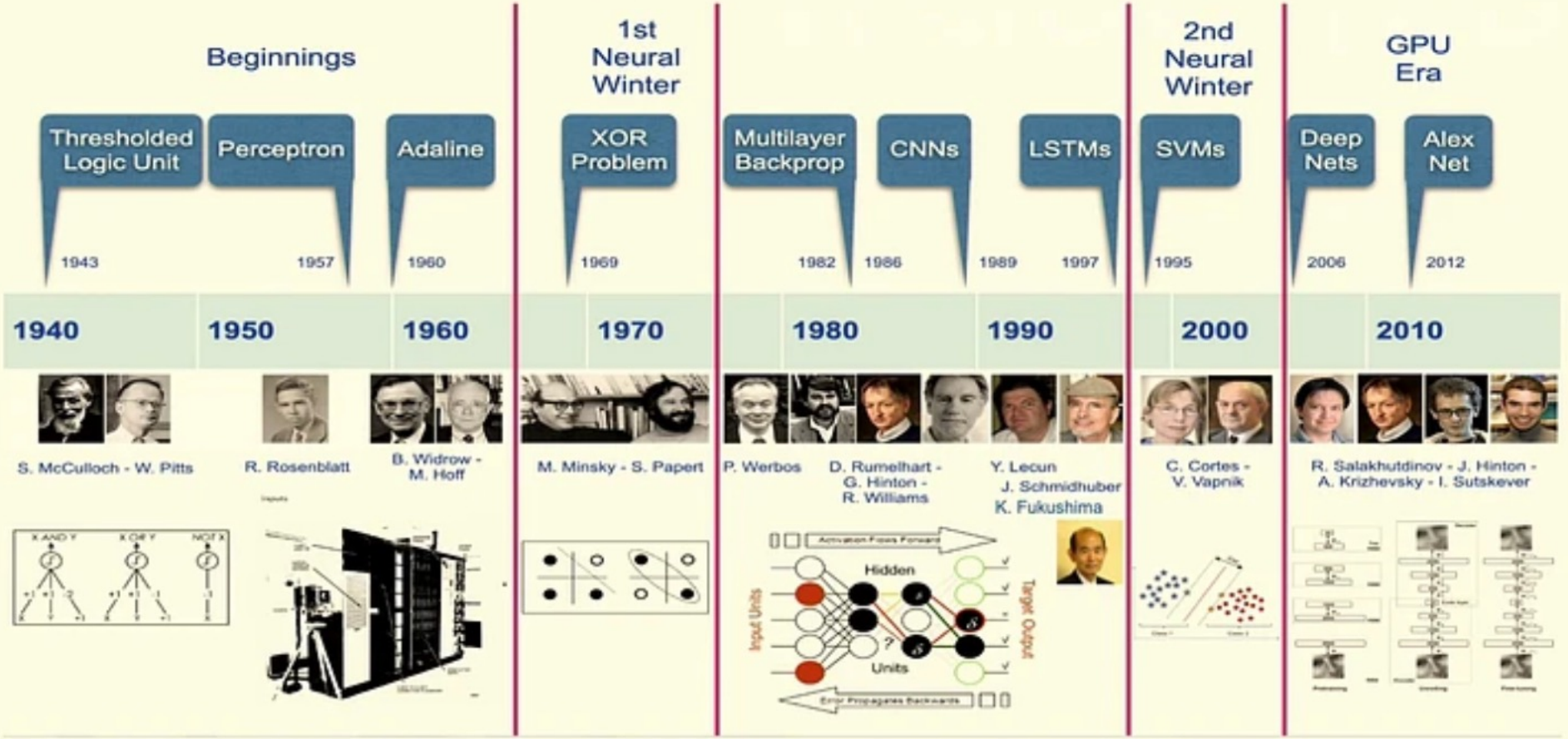
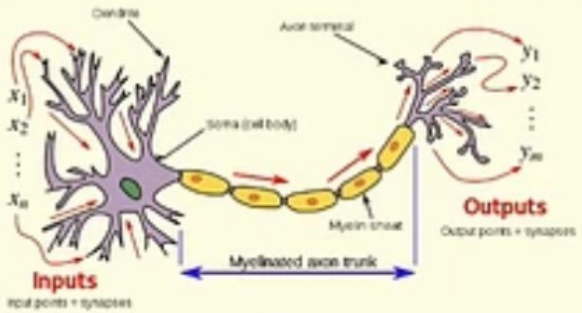
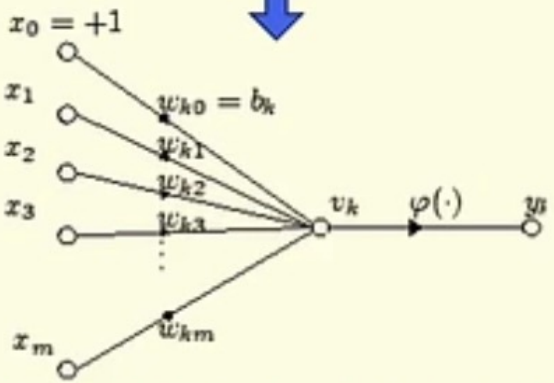


Figure courtesy of Professor Rene Vidal

Artificial Neurons and Neural Networks: Learn from Nature

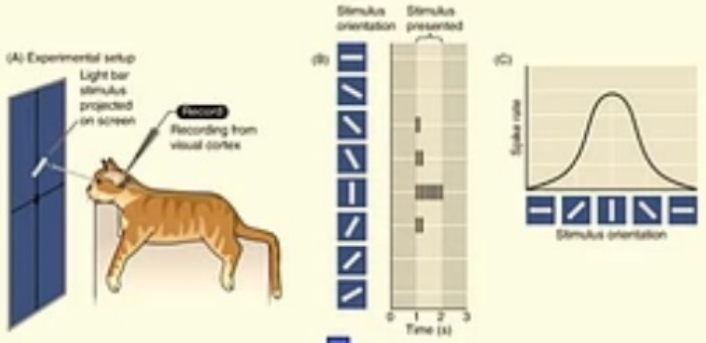
Golgi and Cajal 1888 (1901 Nobel Prize)

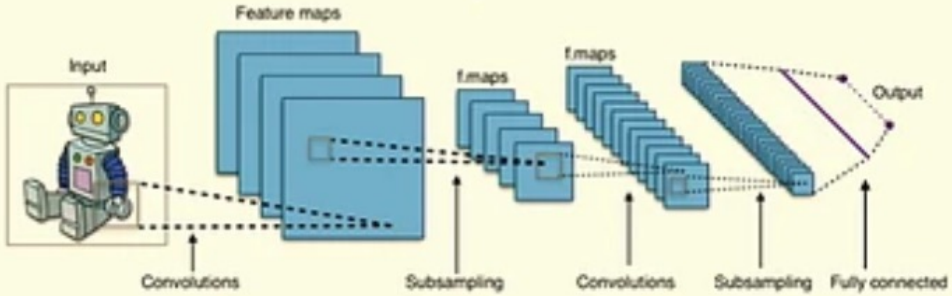




Warren McCulloch & Walter Pitts 1948

Hubel and Wiesel 1959 (1981 Nobel Prize)

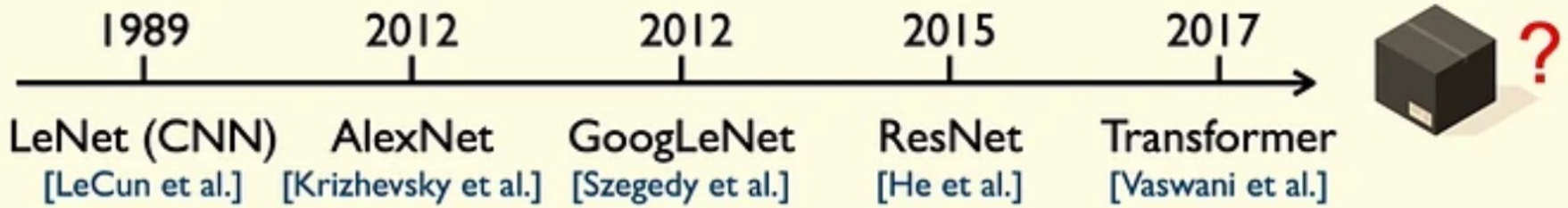


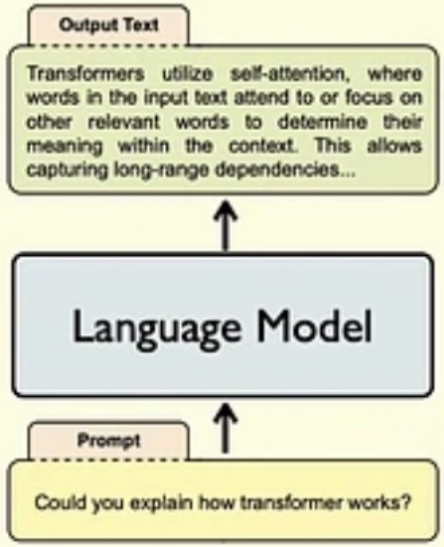


Fukushima 1980 & LeCun 1989 (Turing Award)

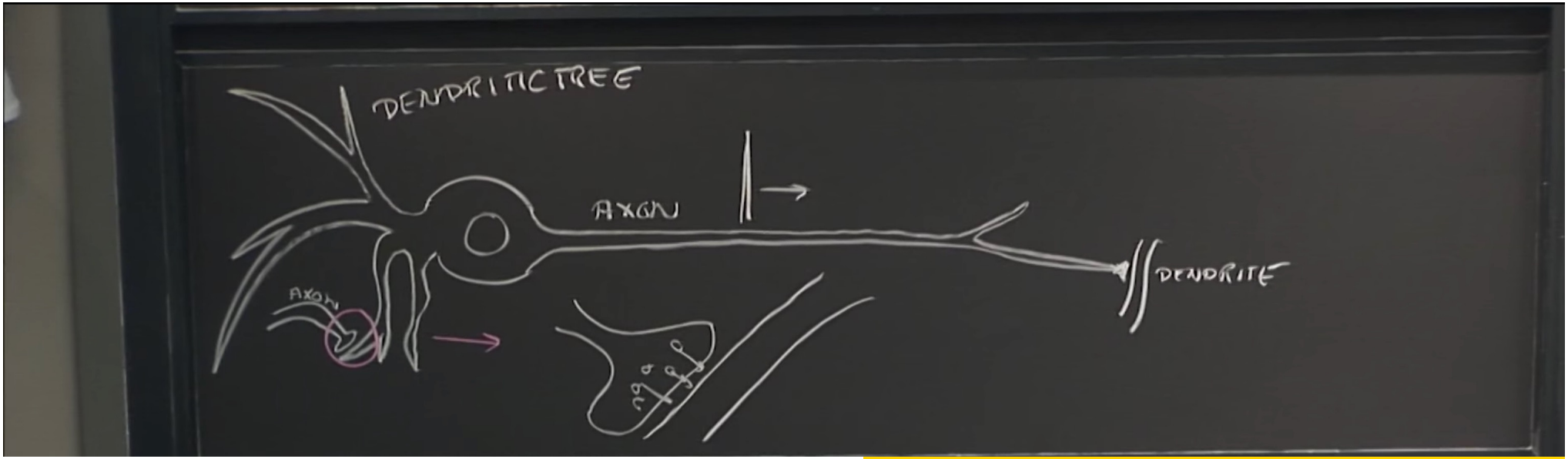
Modern Evolution of Deep Neural Networks

Deep Neural Nets



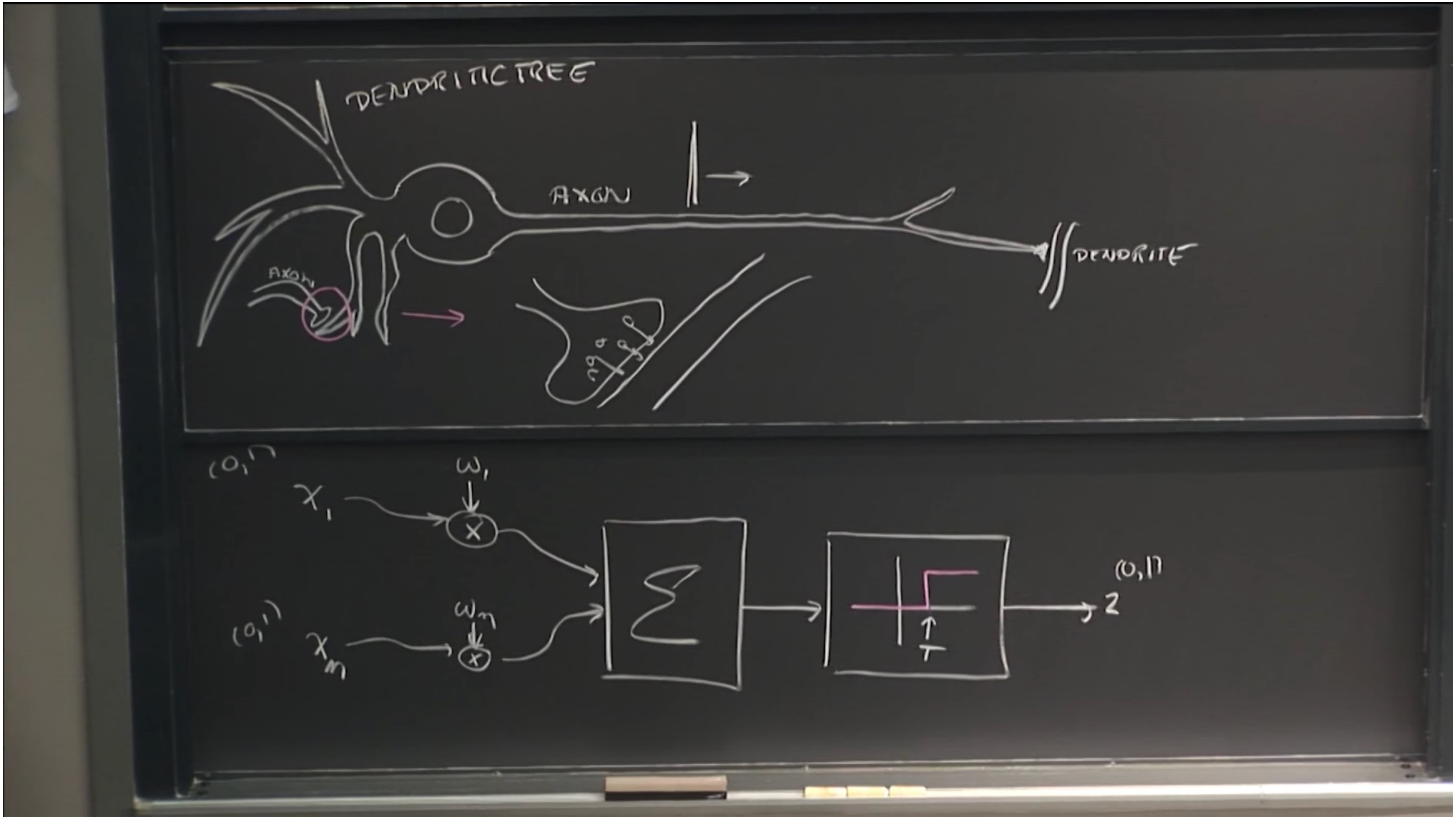


(2024 Nobel Prize Chemistry)



The working of natural neural networks ↑ motivated models for artificial neural networks.

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-12a-neural-nets/

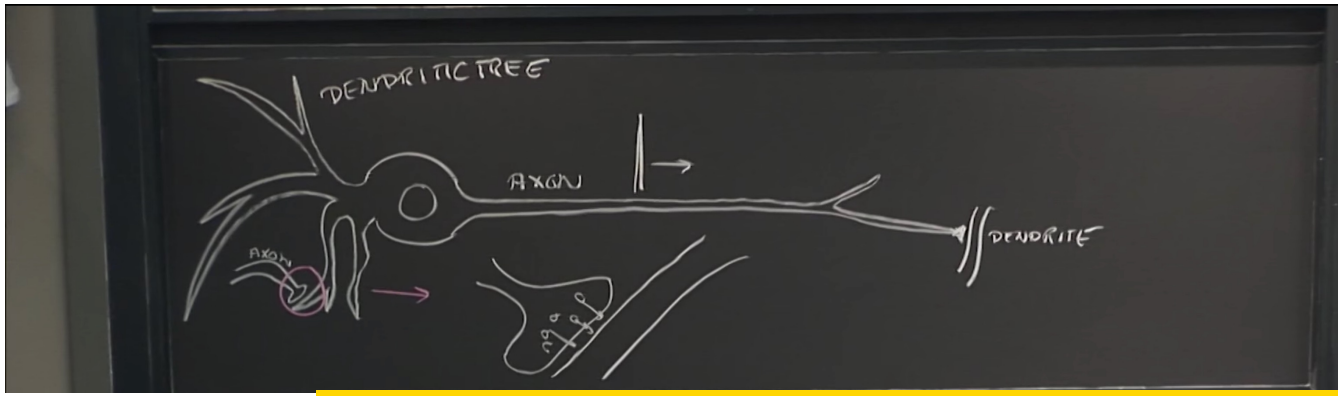


https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-12a-neural-nets/

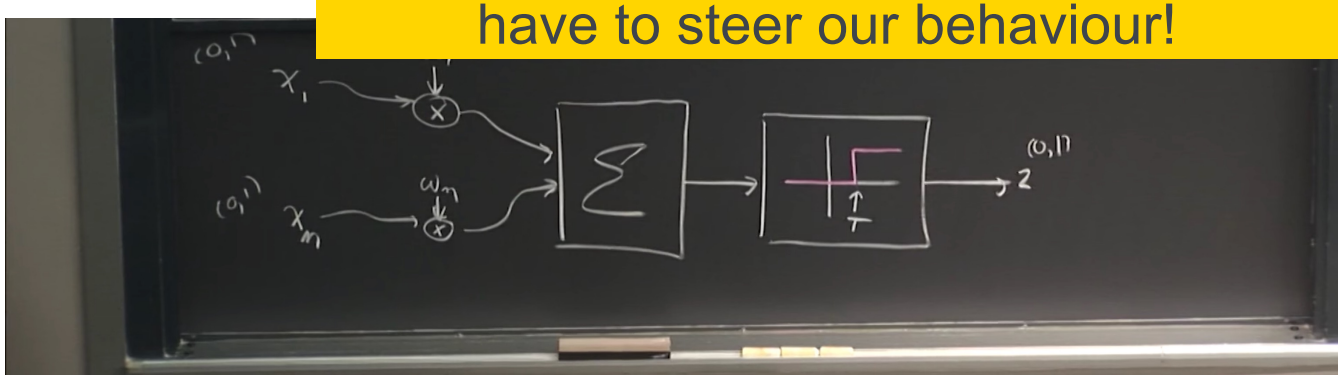
Differences between biological and (typical) artificial neural networks

- Biological neuron
  - potential encoding
    - 100ms scale
  - temporal encoding
    - temporal coding
      - 1ms scale
    - differential equations

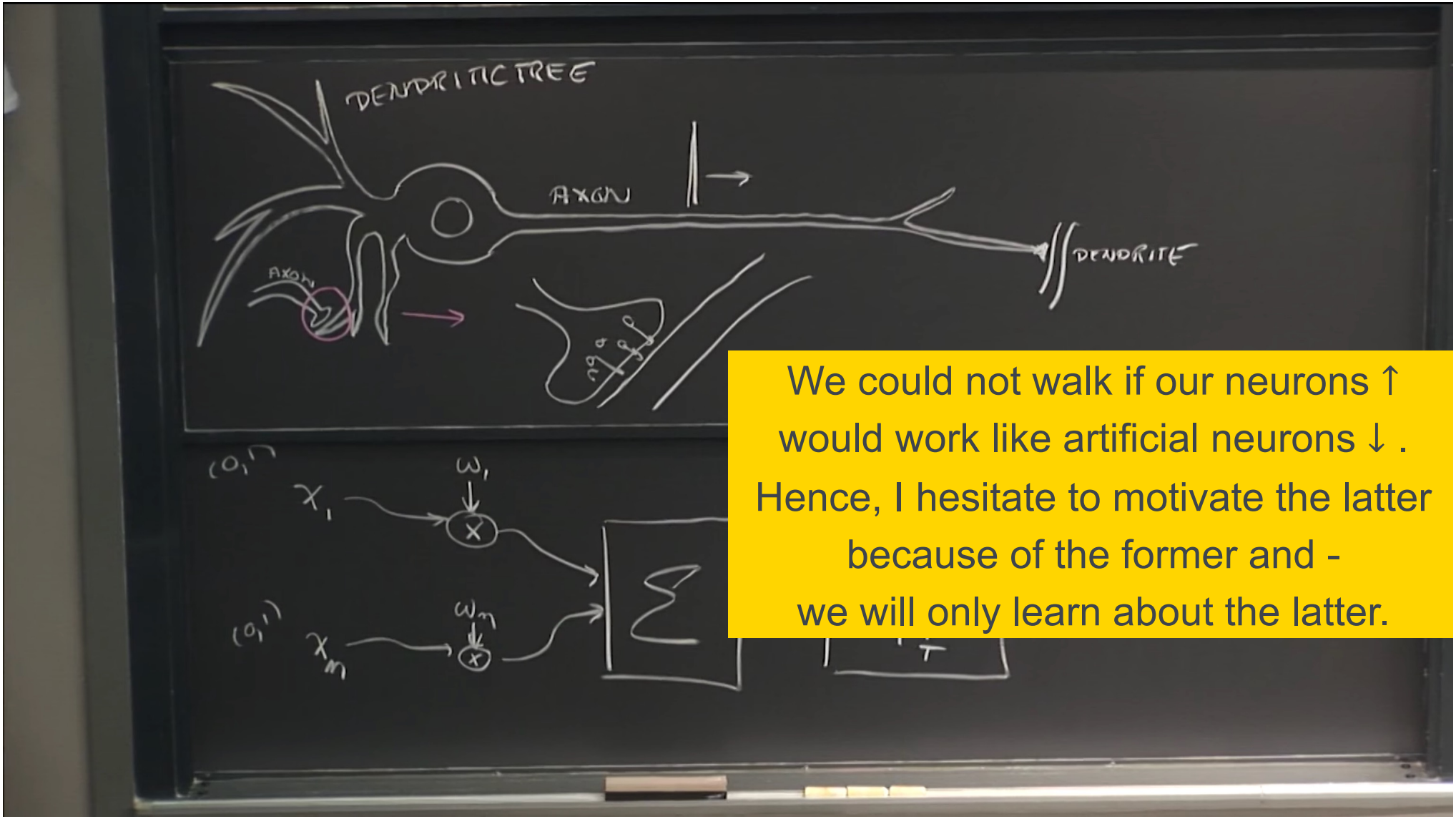
    ~ Spiking neural network
- Artificial neural network
  (as typically used in ML)
  - non-linear statistical models



We could not walk if this encoding would have to steer our behaviour!

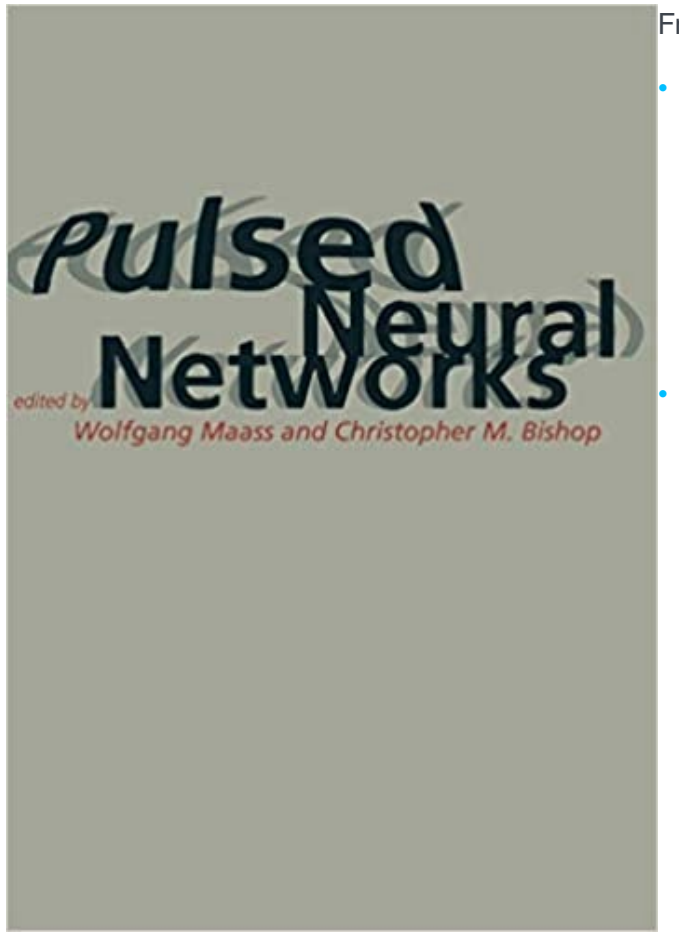


https://en.wikipedia.org/wiki/Neural_coding



https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-12a-neural-nets/

Here, a book from 1999



From the book cover:

- Most practical applications of artificial neural networks are based on a computational model involving the propagation of continuous variables from one processing unit to the next. In recent years, data from neurobiological experiments have made it increasingly clear that biological neural networks, which communicate through pulses, **use the timing of the pulses to transmit information and perform computation**. This realization has stimulated significant research on pulsed neural networks, including theoretical analyses and model development, neurobiological modeling, and hardware implementation.
- This book presents the complete spectrum of current research in pulsed neural networks and includes the most important work from many of the key scientists in the field. Terrence J. Sejnowski's foreword, "Neural Pulse Coding," presents an overview of the topic. The first half of the book consists of longer tutorial articles spanning neurobiology, theory, algorithms, and hardware. The second half contains a larger number of shorter research chapters that present more advanced concepts. The contributors use consistent notation and terminology throughout the book.

Unfortunately, pulsed neural networks have not been a success in technical applications until today. They are also not much researched (in comparison).

Neuromorphic computing

- IBM True North Chip
  https://en.wikipedia.org/wiki/Cognitive_computer#IBM_True_North_chip
- …
- TU Dresden SPINNAKER2 Chip
  (SPIking Neural Network Architecture)
  Low power consumption!
- Liquid neural networks

Liquid neural networks

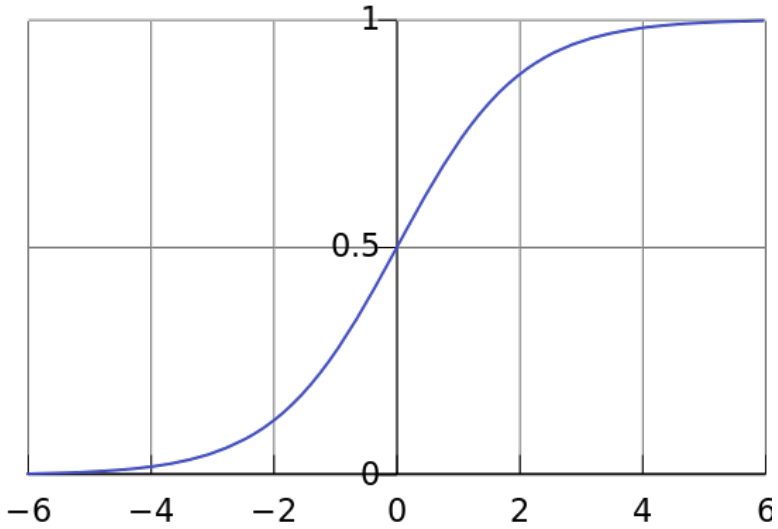
- Simulating C. Elegans
- Using differential equations
- https://www.quantamagazine.org/researchers-discover-a-more-flexible-approach-to-machine-learning-20230207/?fbclid=IwAR3qnPFC-EdF9jzMNu4XubABaM_S9HTKaia8ZjtTPM-xuRqyQLAyNlM2r8o

2 Reminder: Logistic regression

Logistic regression

Model

- Composed of
  - Linear function: $z = x^T\beta$
  - Non-linear **activation function**: $y = \frac{e^z}{1+e^z}$
- Together: $\hat{f}(x) = \frac{e^{x^T\beta}}{1+e^{x^T\beta}}$



Loss function

- Cross entropy and regularization

$$\text{loss}^{\text{logistic}}(\beta) = \sum_{i=1}^{N} H(\bar{y}_i, P(\cdot\,|X = x_i)) + \lambda\,\|\beta\|^2$$

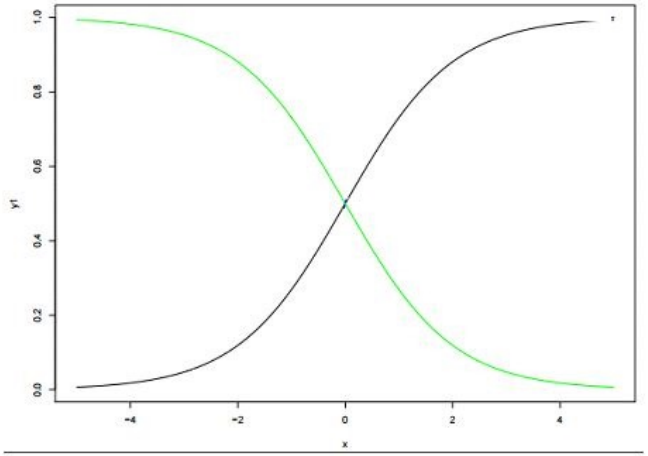
Solving

- adapting the weights, e.g. by gradient descent

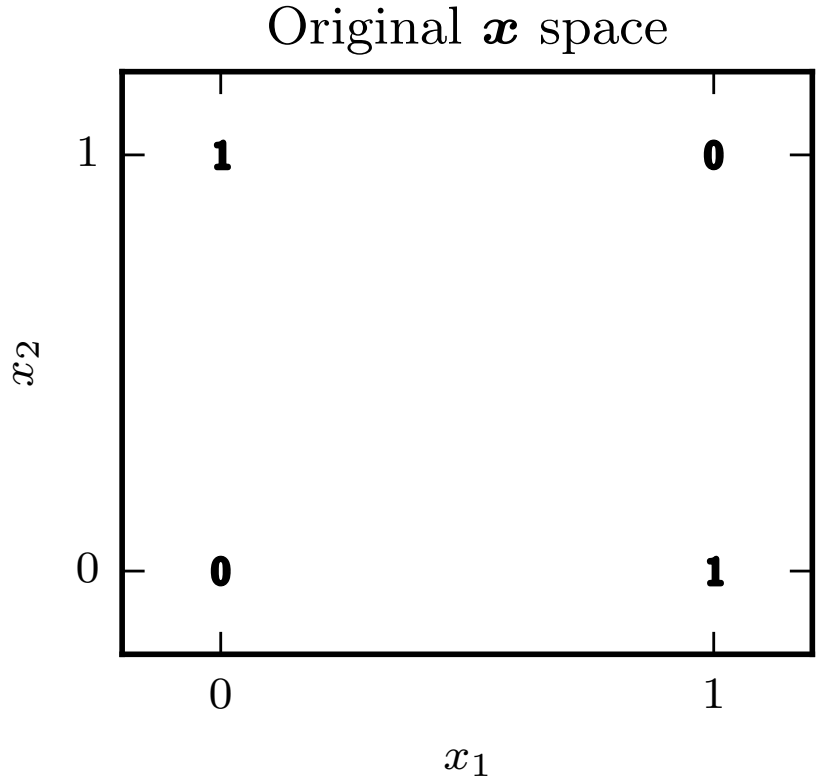
Logistic regression for binary case

- Predict conditional probabilities $\in [0,1]$
  - 2-class example
    - $P(G = 1|X = x) = \frac{e^{x^T\beta}}{1+e^{x^T\beta}}$
    - $P(G = 2|X = x) = \frac{1}{1+e^{x^T\beta}}$

Sum is 1



Problem: Learning XOR with logistic regression



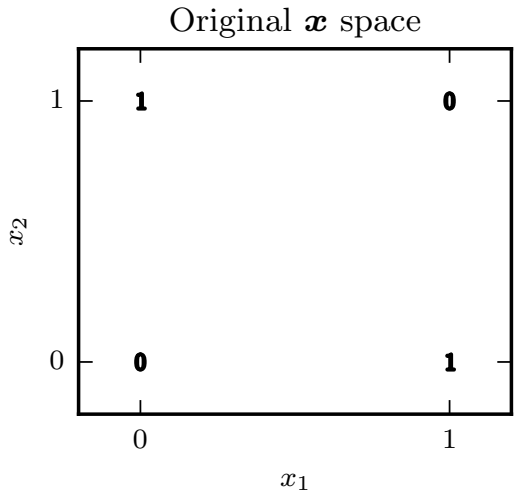
(Goodfellow 2017)

Decision boundary is a hyperplane: Cannot learn XOR

$$\log\frac{P(G=1|X=x)}{P(G=2|X=x)} = \log\frac{\frac{e^{x^T\beta}}{1+e^{x^T\beta}}}{\frac{1}{1+e^{x^T\beta}}} = \log e^{x^T\beta} = x^T\beta$$

Dealing with non-linear separability

1. Kernelization - also works for logistic regression!
2. Composition of non-linear functions



3 Feed-Forward Networks and Activation Functions

Networks: Composing functions

Three layer network:

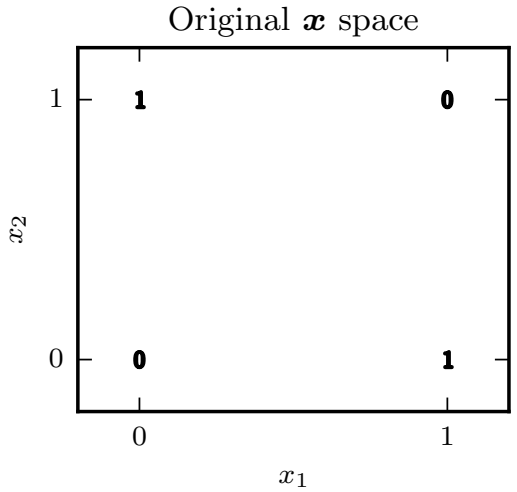
$$f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right)$$

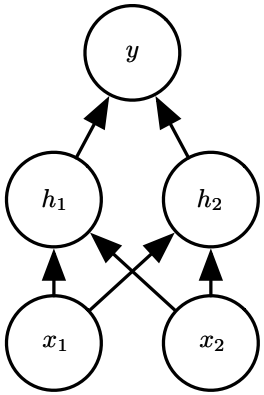
First layer: $f^{(1)}$

Second layer: $f^{(2)}$

Last layer, output layer: $f^{(3)} = \hat{y}$

What about composition of linear functions?





Find solution

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^2(f^1(\boldsymbol{x}))^T$$

With

$$f^1(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) = \boldsymbol{h}$$

$$f^2(\boldsymbol{h}; \boldsymbol{w}, b) = y$$

Then

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{W}^T \boldsymbol{x}$$

A composition of linear functions gives a linear function

→ cannot solve XOR learning problem

One Layer in a Feed-Forward Network

A linear combination of non-linear basis functions (features):

$$f^{(i)}(x) = w_{i,0} + \sum_{j=1}^{l_i} w_{i,j} \cdot \phi(x; \theta_i) = \phi(x; \theta)^T w_i$$

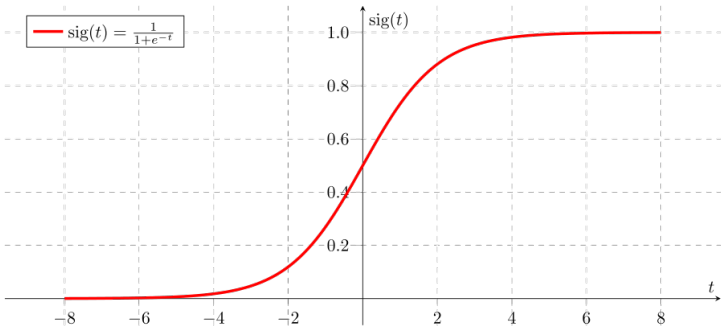
with l_i being **width** of layer i.

How to pick $\phi(x; \theta_i)$?

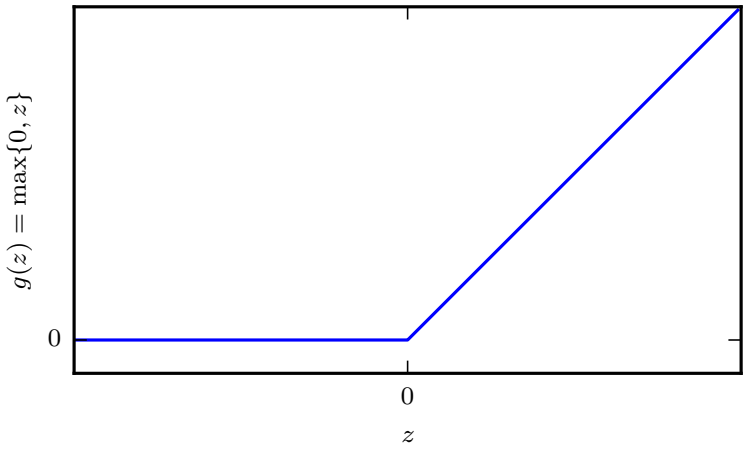
- Core idea: $\phi(x; \theta_i) = g(x^T \theta_i)$
  - learnable linear component: $x^T \theta_i$
  - fixed non-linear activation function: $g(z)$

Activation Functions

Sigmoid (cf logistic regression)



ReLU – Rectified Linear Unit



most used today

Sigmoid: $g(z) = \sigma(z) = \frac{e^z}{1+e^z}$

ReLU: $g(z) = \max\{0, z\}$

Leaky ReLU: $g_\alpha(z) = \max\{\alpha z, z\}$,
(never saturates fully) e.g. $\alpha = 0.1$

Maxout unit: $g(z) = \max_j z_j$

(largest value of group)

Absolute value rectification: $g(z) = |z|$
(e.g. for object recognition)

Hyperbolic tangent: $g(z) = \tanh(z)$

Radial Basis Function: $g(z) = e^{-\frac{1}{2}\|z-c_j\|^2}$

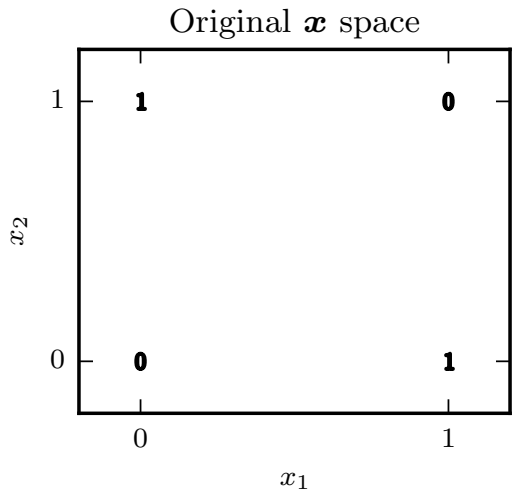
+ Simple
+ Efficient to compute with
+ Non-linear
- Not differentiable at 0

Possible Solution to the XOR problem

$$\hat{f}\,(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b) = \boldsymbol{w}^T \max\{\boldsymbol{0},\boldsymbol{W}^T\boldsymbol{x}+\boldsymbol{c}\} + b$$

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$Y^T = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



Possible Solution to the XOR problem

$$\hat{f}\,(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b) = \boldsymbol{w}^{T}\max\{\boldsymbol{0},\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x}+\boldsymbol{c}\}+b$$

$$W = \begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix} \qquad \boldsymbol{X} = \begin{bmatrix}0 & 0 & 1 & 1\\ 0 & 1 & 0 & 1\end{bmatrix} \qquad \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{X} = \begin{bmatrix}0 & 1 & 1 & 2\\ 0 & 1 & 1 & 2\end{bmatrix}$$

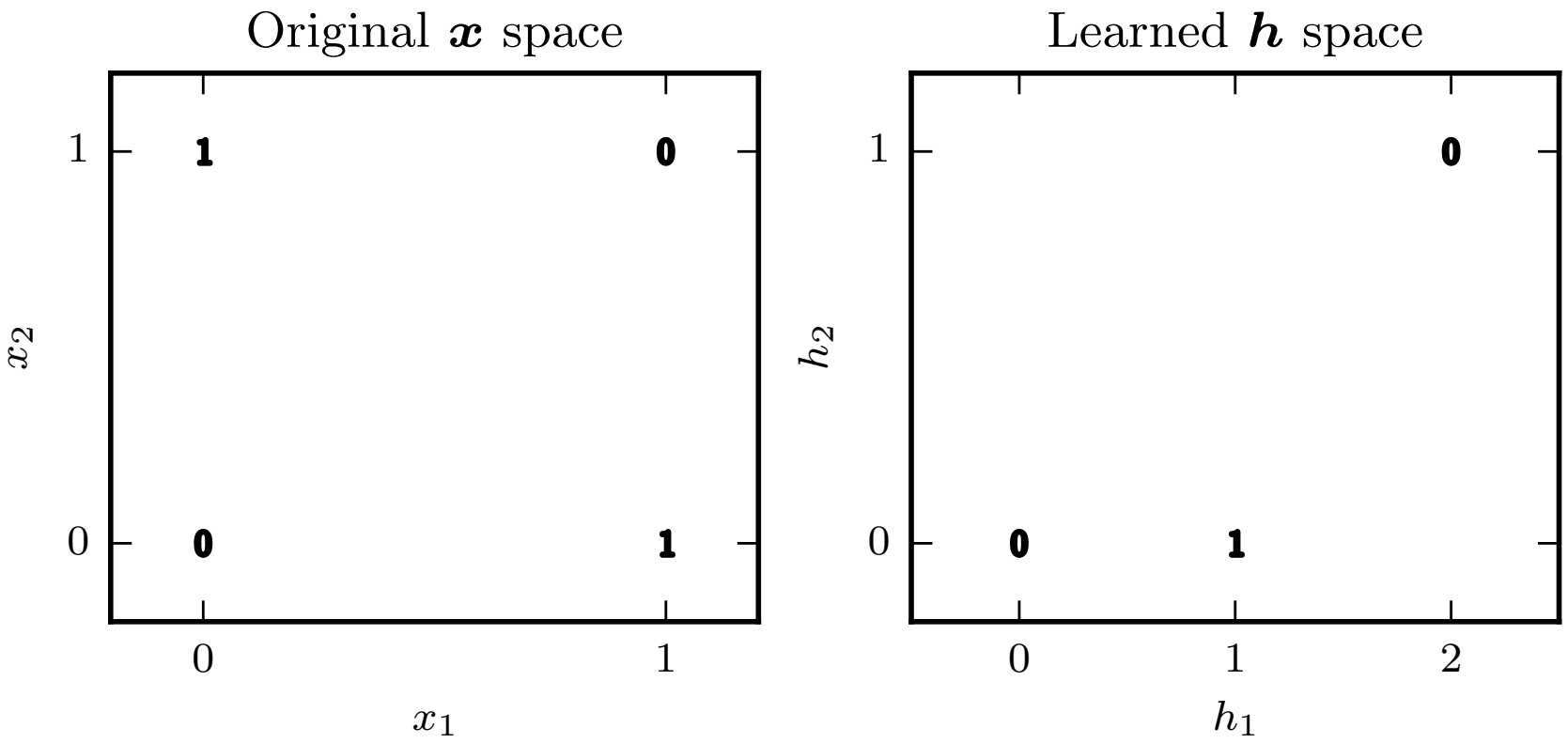
$$c = \begin{bmatrix}0\\ -1\end{bmatrix} \qquad \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{X} + [\boldsymbol{c} \quad \boldsymbol{c} \quad \boldsymbol{c} \quad \boldsymbol{c}] = \begin{bmatrix}0 & 1 & 1 & 2\\ -1 & 0 & 0 & 1\end{bmatrix}$$

RELU (pointwise)

$$\begin{bmatrix}0 & 1 & 1 & 2\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$w = \begin{bmatrix}1\\ -2\end{bmatrix} \qquad [1 \quad -2]\begin{bmatrix}0 & 1 & 1 & 2\\ 0 & 0 & 0 & 1\end{bmatrix} = [0 \quad 1 \quad 1 \quad 0]$$

Solving XOR problem



Universal approximation theorem

Theorem: Let $\varphi: \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and monotonically-increasing continuous function.

Let C denote the space of continuous function on $[0,1]^m$.

Given **any** $\varepsilon > 0$ and **any** function $f \in C$,

there exist $N \in \mathbb{N}$, $v_i, b_i \in \mathbb{R}, w_i \in \mathbb{R}^m$, with $i = 1, \dots, N$,
such that

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \, \varphi(\boldsymbol{w_i^T x} + b_i)$$

and

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in [0,1]^m$. □

On compact spaces, simple, sufficiently broad neural networks with one type of non-linear activation function approximate any continuous function arbitrarily well.
BUT: This is **not** a practical proposition! N might have to be exponential in number of data points.

Summary: One Layer Feed-forward Network

Model

- Composed of
  - Linear function: $\mathrm{z} = x^T W + c$
  - Non-linear **activation function** $\boldsymbol{g(z)}$:

$$g(z) = \mathrm{ReLU}(z), g(z) = \sigma(z) = \frac{e^z}{1+e^z}, g(z) = \tanh(z), g(z) = e^{-\frac{1}{2}\|z-c_j\|^2}, \ldots$$

- Together: $\hat{f}(x) = w^T g(x^T W + c) + b$
  - If we capture the „bias unit“ and write $x^T = (1\ x_1 \ldots x_d)$, we can (again) simplify this to $\hat{f}(x) = w^T g(x^T W)$

Loss function ??

Solving ???

Chain-based architecture (fully connected)

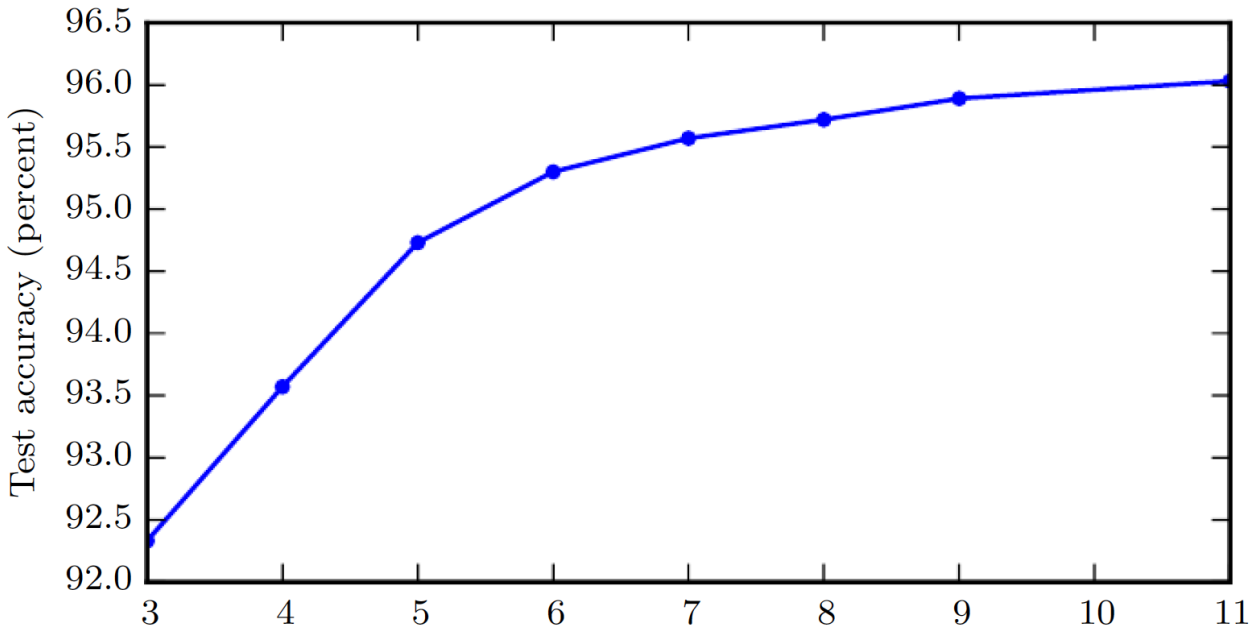
- Input layer (not counted): $x \in \mathbb{R}^{d+1}, x = h^{(0)}$
- First layer: $h^{(1)} = g^{(1)}\left(W^{(1)T}x + b^{(1)}\right)$
- Second layer: $h^{(2)} = g^{(2)}\left(W^{(2)T}h^{(1)} + b^{(2)}\right)$
- ...
- n-th layer: $h^{(n)} = g^{(n)}\left(W^{(n)T}h^{(n-1)} + b^{(n)}\right) = \hat{y}$

Design considerations

- Depth of network
- Width of network l_i at each layer i
- $W^i \in \mathbb{R}^{l_i} \times \mathbb{R}^{l_{i-1}}$

Effect of depth

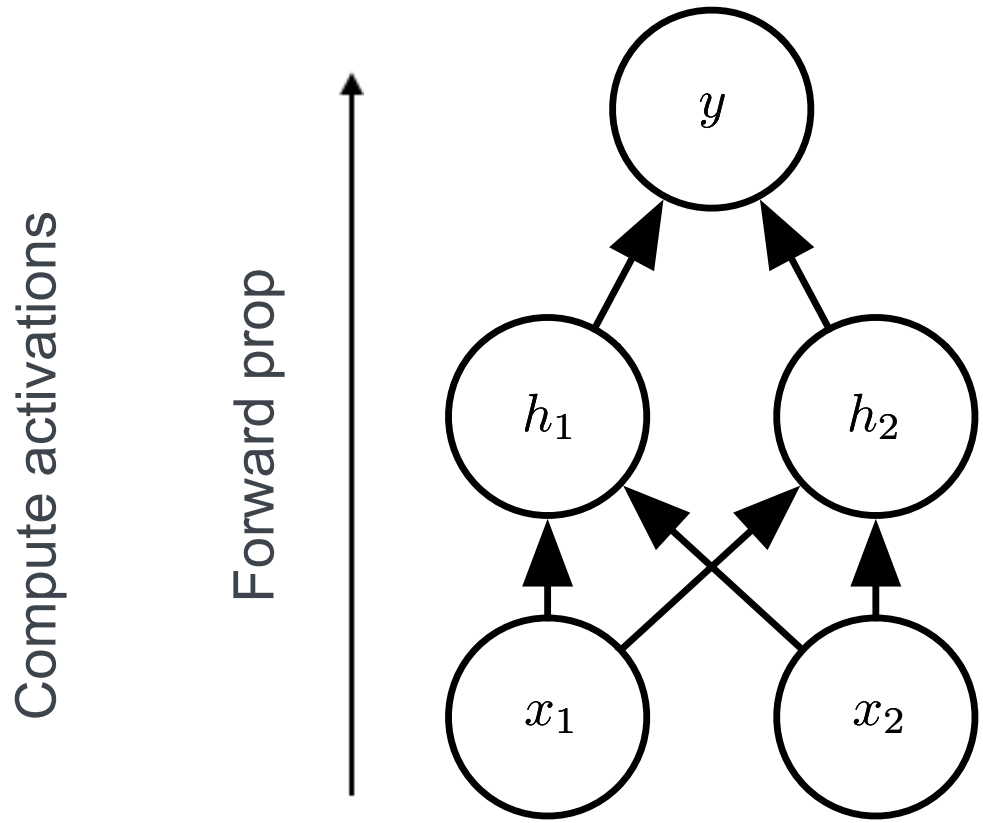
Experiment on recognizing multidigit numbers with increasing depth (convolutional* with 1 or 2 fully connected layers)



*Convolutional network to be explained in next lecture

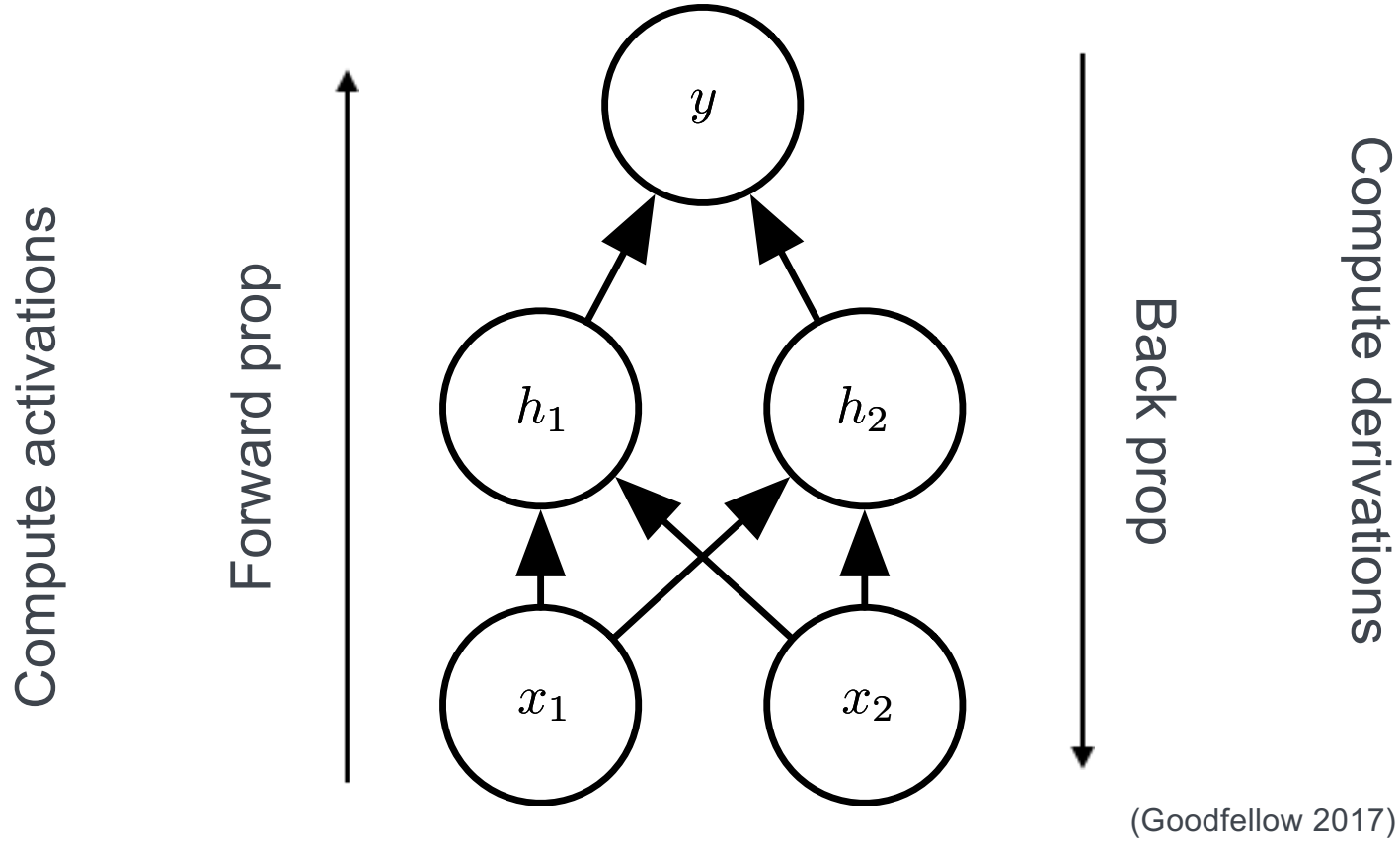
4 Backpropagation

Forward Propagation



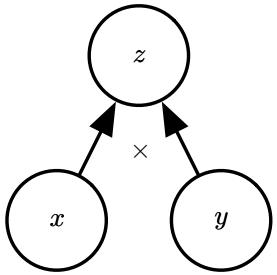
Back-Propagation

Compute loss L($\boldsymbol{\theta}$) considering $\boldsymbol{y}$ (data) vs $\hat{\boldsymbol{y}}$ (output activation/prediction)



(Goodfellow 2017)

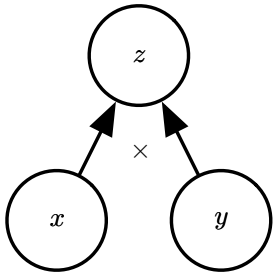
Language for representing the backprop problem: Computation Graphs



(a)

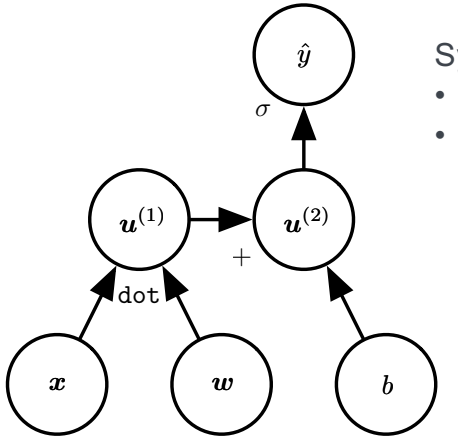
Multiplication

Language for representing the backprop problem: Computation Graphs



(a)

Multiplication



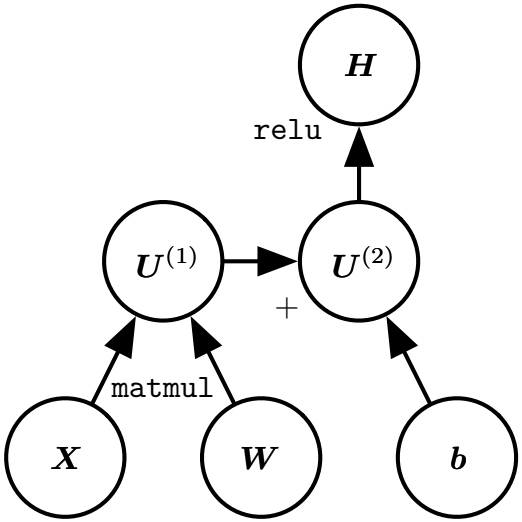
(b)

Logistic regression

Syntax elements:

- **Nodes** represent **variables**
- (multiple) edges with labels represent **operations**

Language for representing the backprop problem: Computation Graphs



(c)

ReLU Layer

Syntax elements:

- **Nodes** represent **variables**
- (multiple) edges with labels represent **operations**

Some notational conventions for computation graph

n many values of nodes in the network

- topologically sorted

Example:

$u_1 \dots u_6$

Input layer with values for k nodes:

$u_1 \dots u_k$

$u_1 = x_1, u_2 = w, u_3 = b$

Set of values of parents of node i:

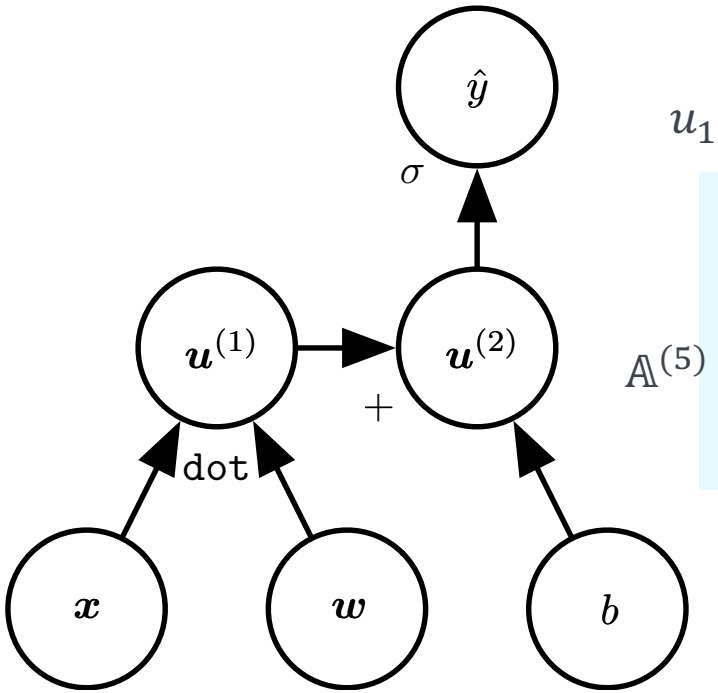
$\mathbb{A}^{(i)}$

$u_4 = u^{(1)}, u_5 = u^{(2)}$

$\mathbb{A}^{(4)} = \{u_1, u_2\}$

$\mathbb{A}^{(5)} = \{u_3, u_4\} = \{b, u^{(1)}\}$

$\mathbb{A}^{(6)} = \{u_5\}$



Operator for node i:

$f^{(i)}$

$f^{(4)} = \text{dot}, f^{(5)} = +$

$f^{(6)} = \sigma$

Output layer with l nodes:

$u_{n-l+1} \dots u_n$

$u_6 = \hat{y}$

Forward propagation in computation graph

n many values of nodes in the network

Input layer with values for k nodes:

$$u_1 \dots u_k$$

Set of values of parents of node i:

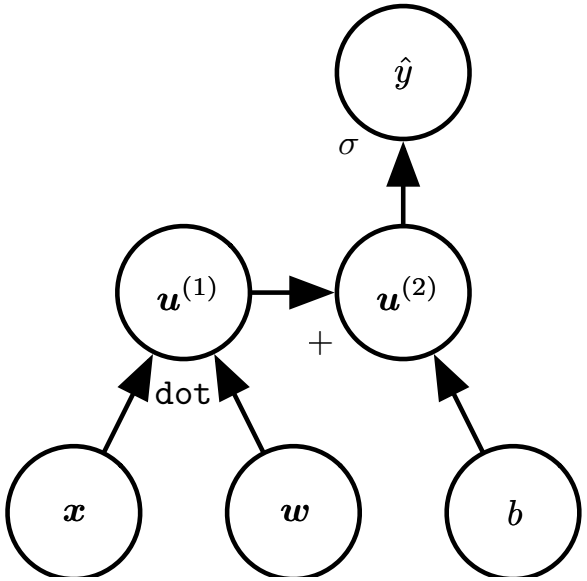
$$\mathbb{A}^{(i)}$$

Operator for node i:

$$f^{(i)}$$

Output layer with l nodes:

$$u_{n-l+1} \dots u_n$$



for $i = k + 1 \dots n$ **do**

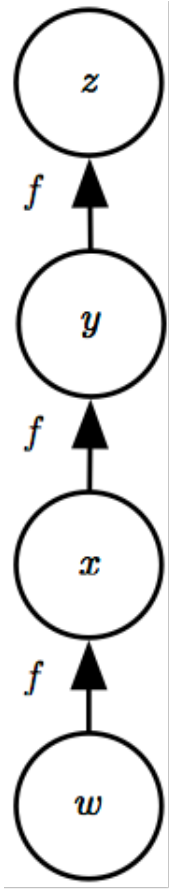
$\mathbb{A}^{(i)} \leftarrow \{u_j \mid j \in \text{parent}(u_i)\}$;

$u_i \leftarrow f^{(i)}(\mathbb{A}^{(i)})$;

end for

return $\mathbf{u}_n$

Some ideas of backpropagation in a simple example



$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.51}$$

$$= f'(y)f'(x)f'(w) \tag{6.52}$$

$$= f'(f(f(w)))f'(f(w))f'(w) \tag{6.53}$$

Back-prop avoids computing this twice

Backpropagation in Tiny Example Computation Graph

Input

$$x = 1, y = 2; \text{target } z = 3$$

Forward result:

$$\hat{z} = 2$$

Increased $\hat{z}$ reduces error

Assume MSE as loss function:

$$J = L(\hat{z}, z) = \frac{1}{2}(\hat{z} - z)^2 = \frac{1}{2}$$

Gradient on loss function L:

$$\nabla_{\hat{z}} J = \nabla_{\hat{z}} L(\hat{z}, z) = 2\frac{1}{2}(\hat{z} - z)$$
$$= (\hat{z} - z) = \hat{z} - 3 = -1$$

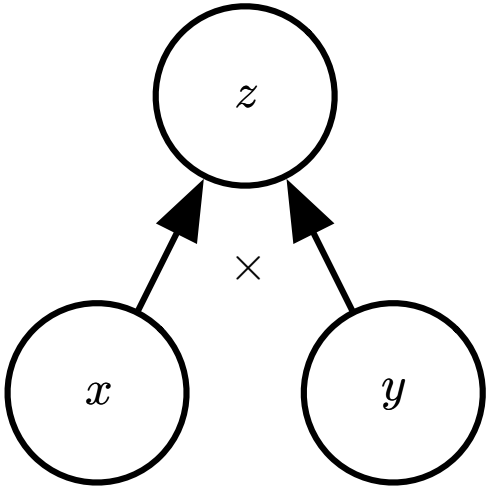
Backpropagation

$$\nabla_{(x,y)} J = \begin{pmatrix} -2 \\ -1 \end{pmatrix}$$

Partial derivatives

$$\nabla_x J = \nabla_{\hat{z}} J \cdot \nabla_x \hat{z} = -1 \cdot \frac{\partial(x \cdot y)}{\partial x} = -y = -2$$

$$\nabla_y J = \nabla_{\hat{z}} J \cdot \nabla_y \hat{z} = -1 \cdot \frac{\partial(x \cdot y)}{\partial y} = -x = -1$$



Conclusion:
If y was a weight it should be increased to bring the error down

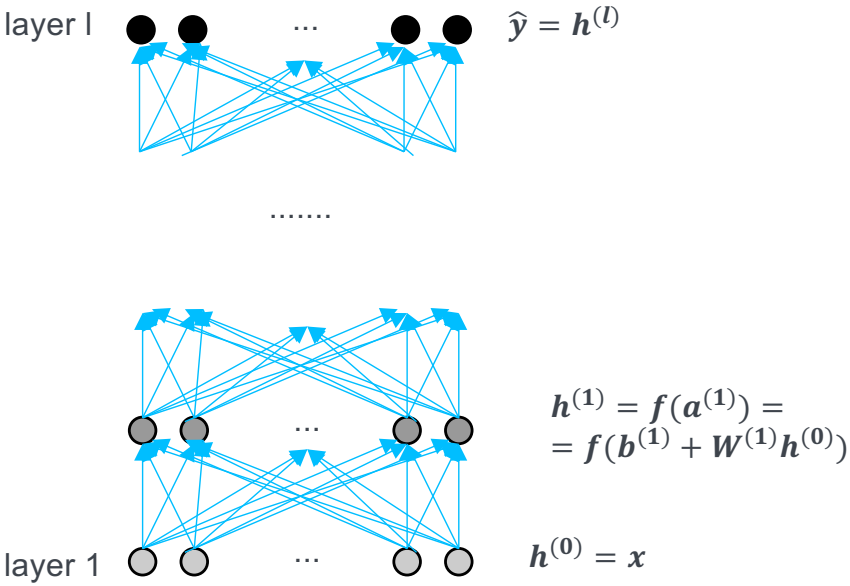
$$y^{new} = y - \eta(-1)$$

Larger Example - Forward: Fully connected feed-forward network

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

regularization



(Goodfellow 2017)

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

regularization

Larger Example - Backward: Fully connected feed-forward network

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

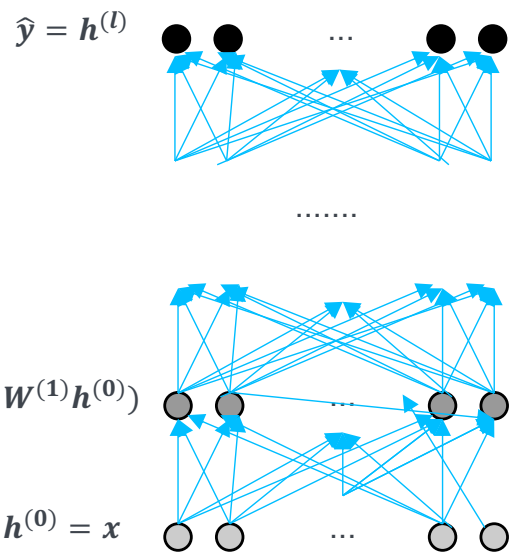
Compute gradients on weights and biases where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$



Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses, in addition to the input $\boldsymbol{x}$, a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

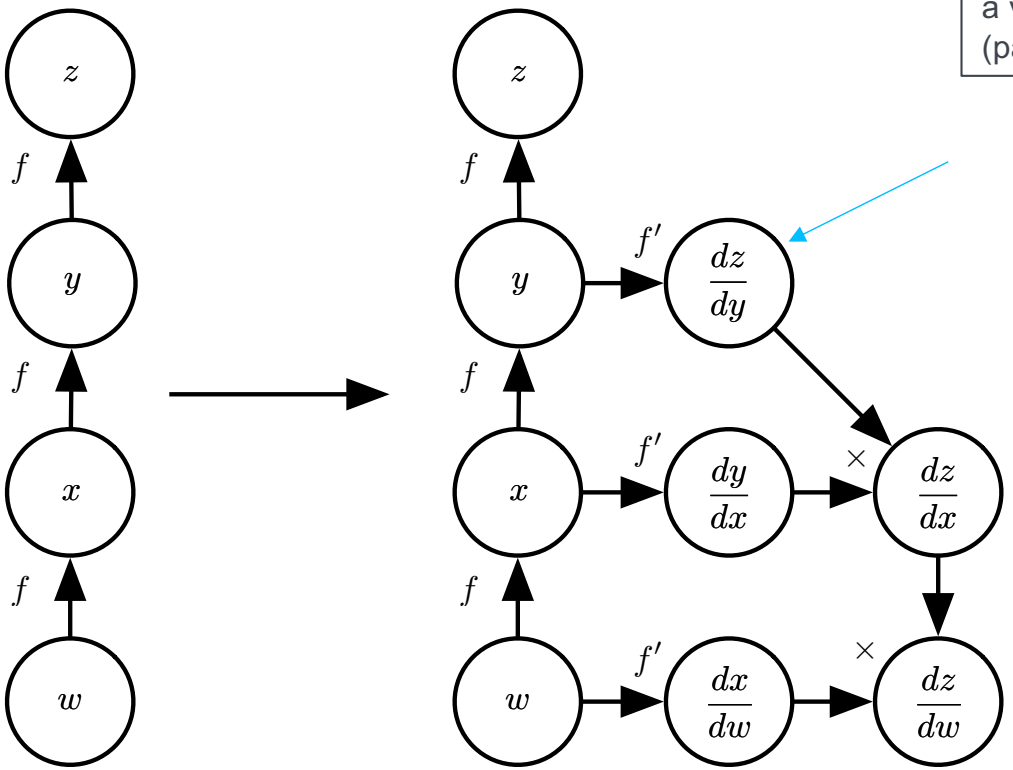
Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

Generalization 1: Symbol-to-Symbol derivatives

For each node representing a value, add a new node for (partial) derivatives



Backpropagation graph

Some derivative nodes might be needed twice, but are computed only once, **but not in this example**

Generalization 2: Avoid double computation

Algorithm 6.6 The inner loop subroutine `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$ of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and `grad_table`
Require: $\mathcal{G}$, the graph to modify
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient
Require: `grad_table`, a data structure mapping nodes to their gradients

if $\mathbf{V}$ is in `grad_table` **then**
  Return `grad_table`$[\mathbf{V}]$
end if
$i \leftarrow 1$
for $\mathbf{C}$ in `get_consumers`$(\mathbf{V}, \mathcal{G}')$ **do**
  `op` $\leftarrow$ `get_operation`$(\mathbf{C})$
  $\mathbf{D} \leftarrow$ `build_grad`$(\mathbf{C}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$
  $\mathbf{G}^{(i)} \leftarrow$ `op.bprop`(`get_inputs`$(\mathbf{C}, \mathcal{G}'), \mathbf{V}, \mathbf{D})$
  $i \leftarrow i + 1$
end for
$\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
`grad_table`$[\mathbf{V}] = \mathbf{G}$
Insert $\mathbf{G}$ and the operations creating it into $\mathcal{G}$
Return $\mathbf{G}$

Dynamic programming

If available, look up, else compute

We will not ask for details of this slide in the exam

5 Example application

The optimization of a deep neural network

- is a rather ill-posed problem,
- has many degrees of freedom (compared to linear models),

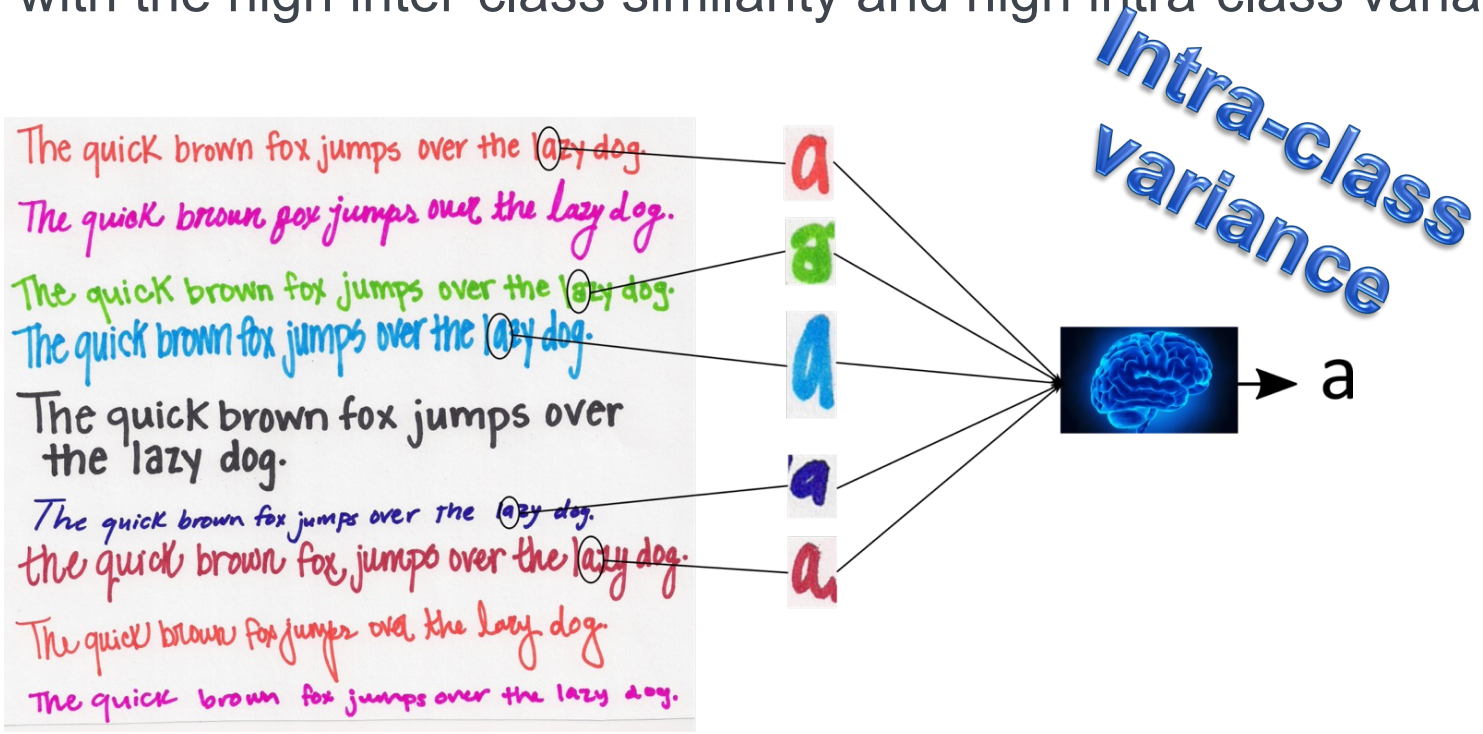
- Deep Neural Network especially useful if there is no previous knowledge about the data generating process

- image
- audio
- text

In these areas deep neural networks outperform other areas by far

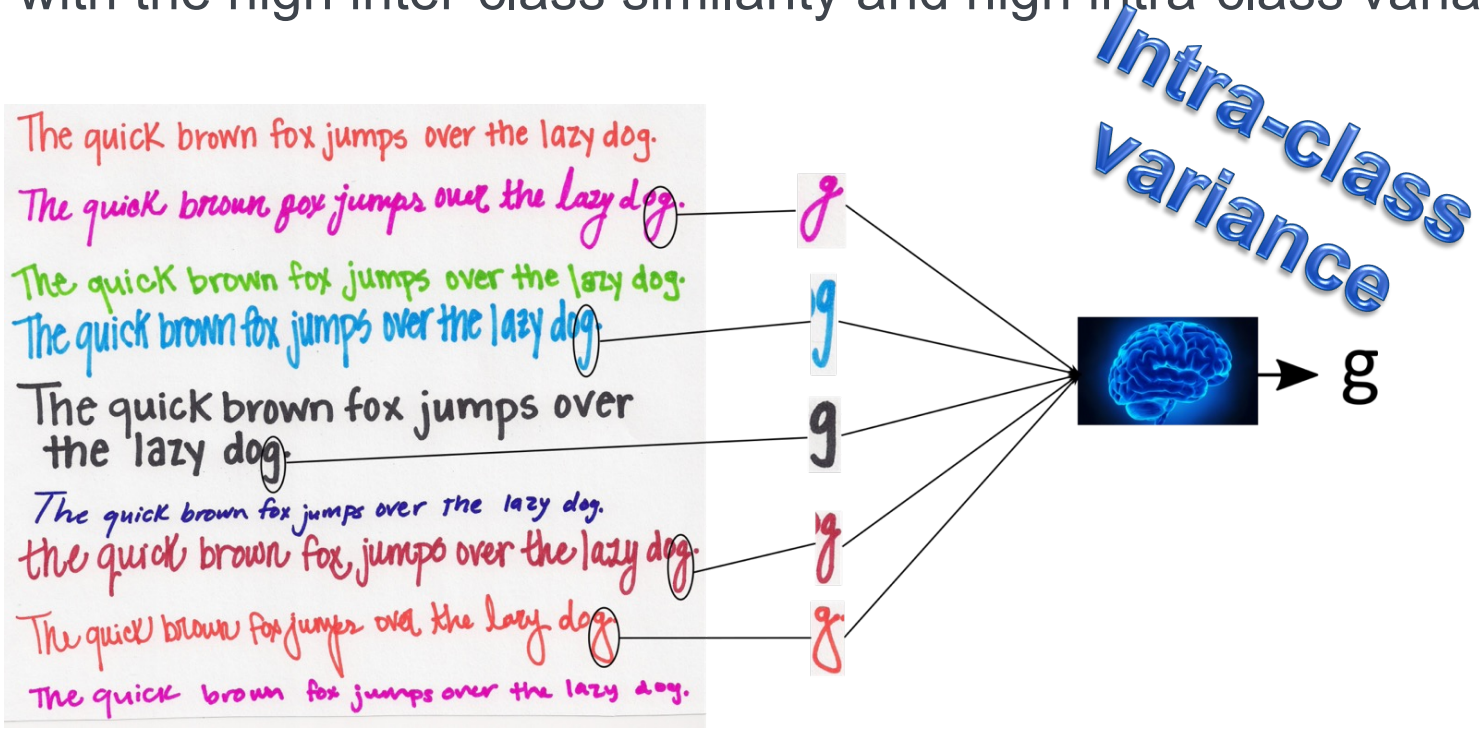
Neural Networks

- Example: Handwritten characters classification
  - It is very simple for human beings to understand handwritten texts, even with the high inter-class similarity and high intra-class variance.



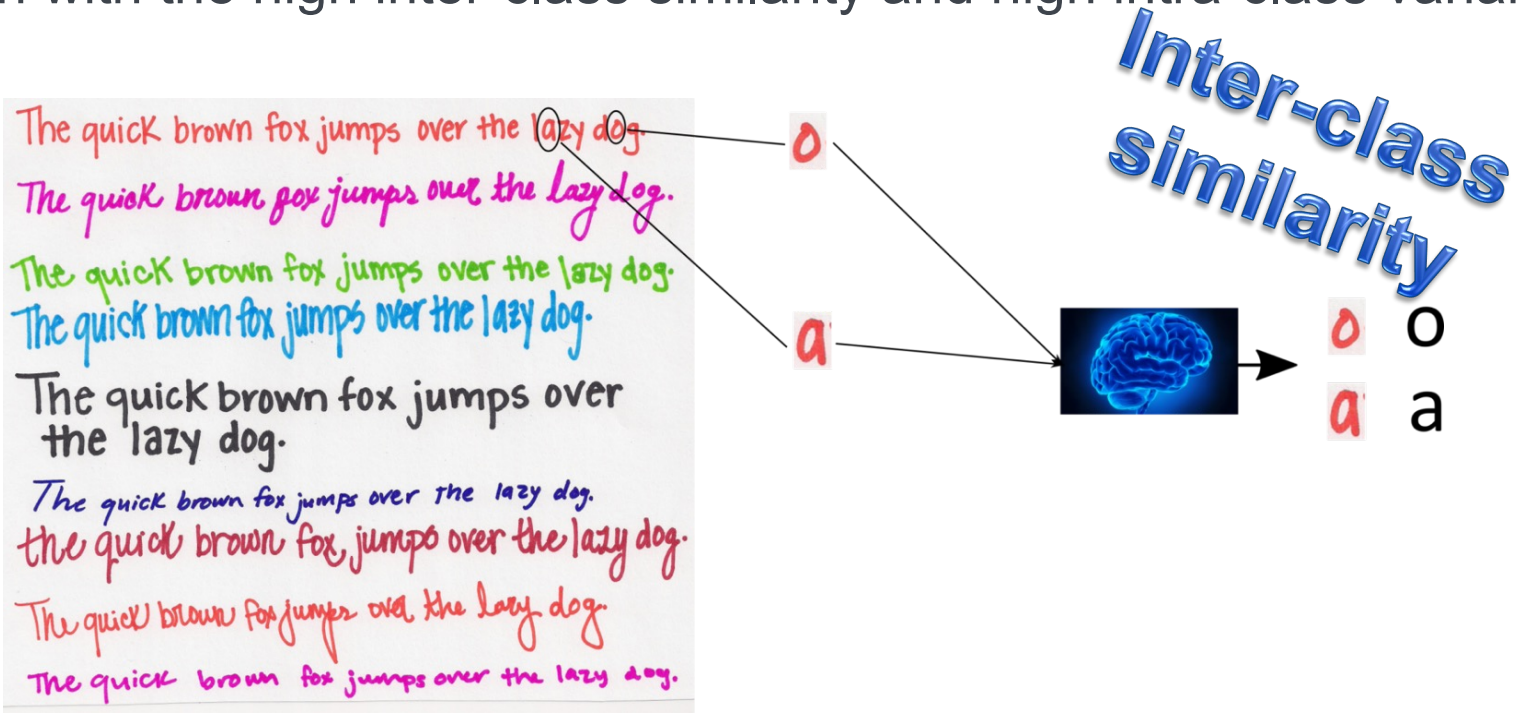
Neural Networks

- Example: Handwritten characters classification
  - It is very simple for human beings to understand handwriting texts, even with the high inter-class similarity and high intra-class variance.



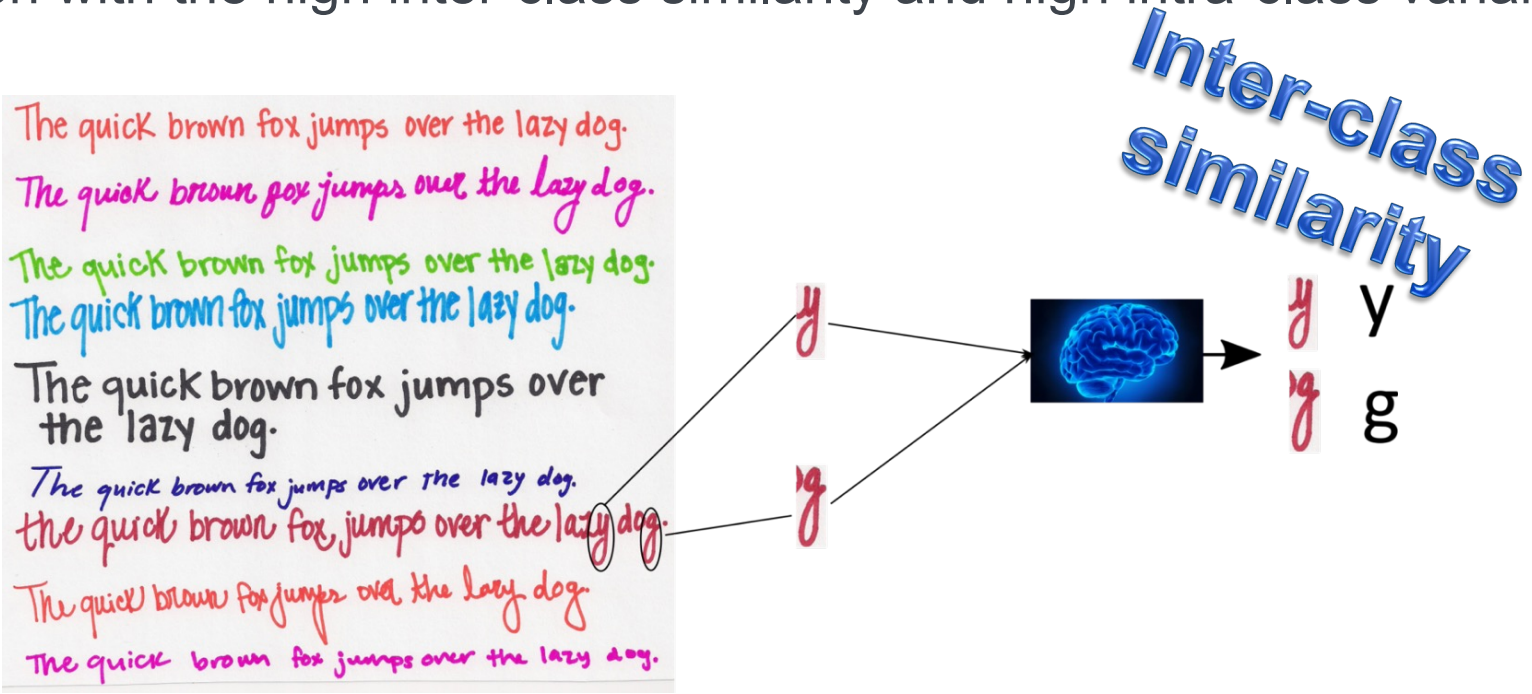
Neural Networks

- Example: Handwritten characters classification
  - It is very simple for human beings to understand handwriting texts, even with the high inter-class similarity and high intra-class variance.

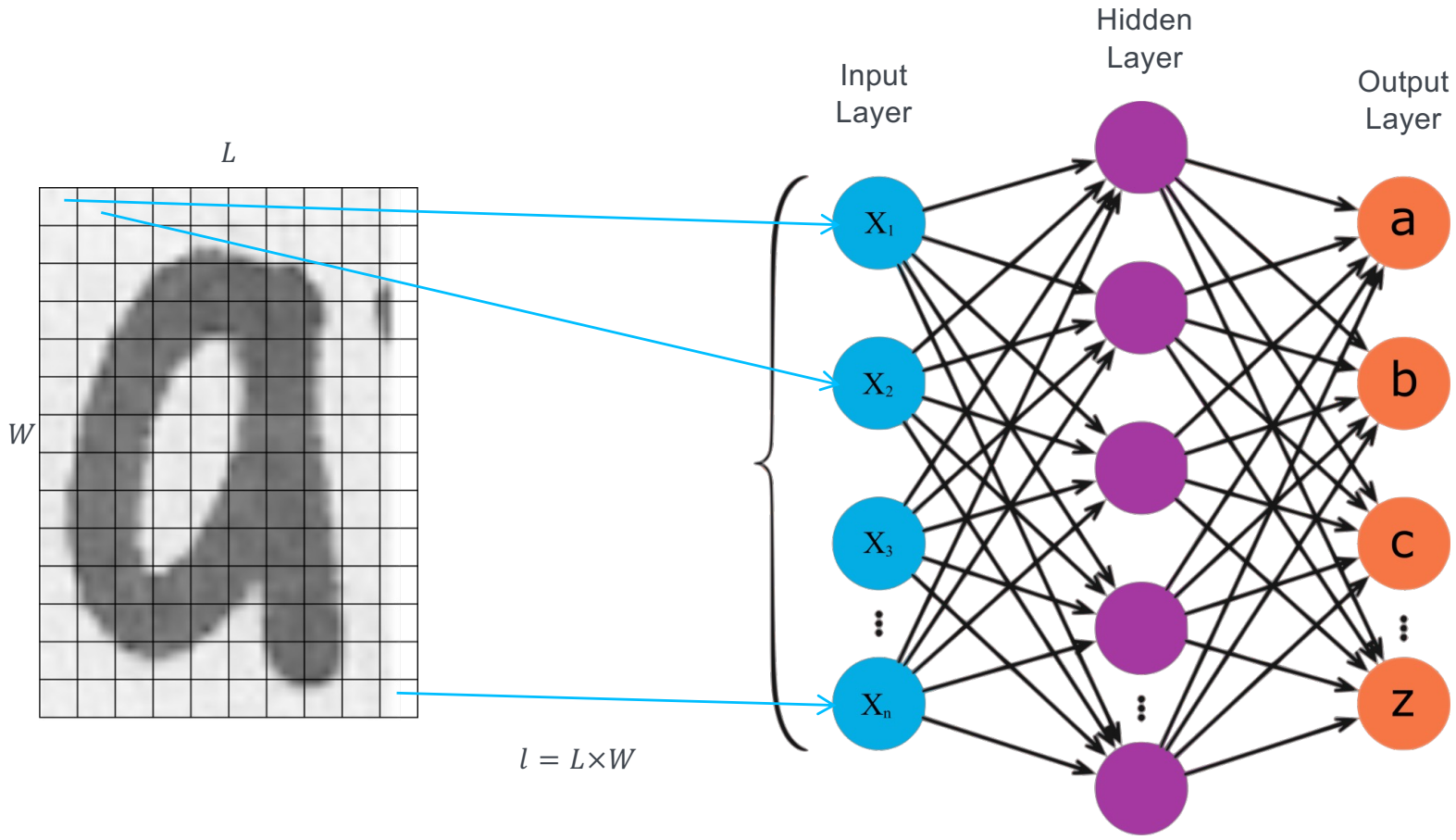


Neural Networks

- Example: Handwritten characters classification
  - It is very simple for human beings to understand handwriting texts, even with the high inter-class similarity and high intra-class variance.

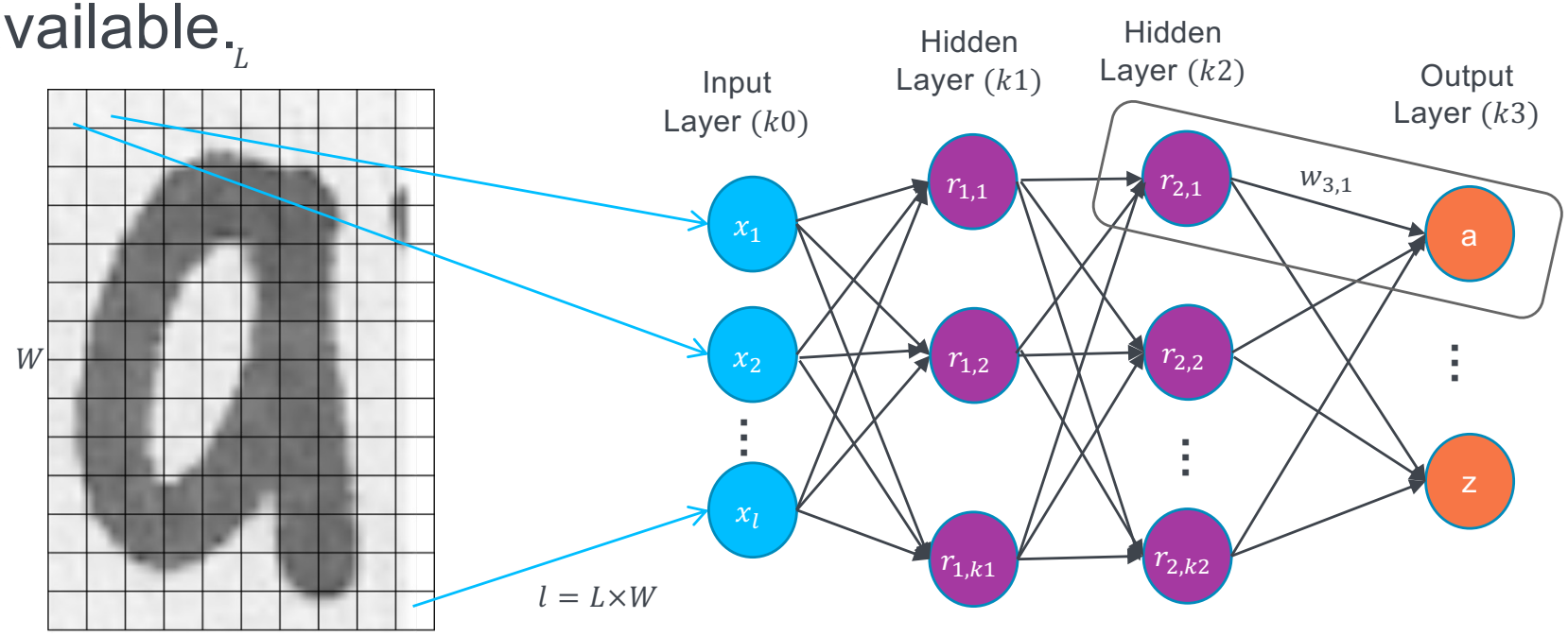


Neural Networks

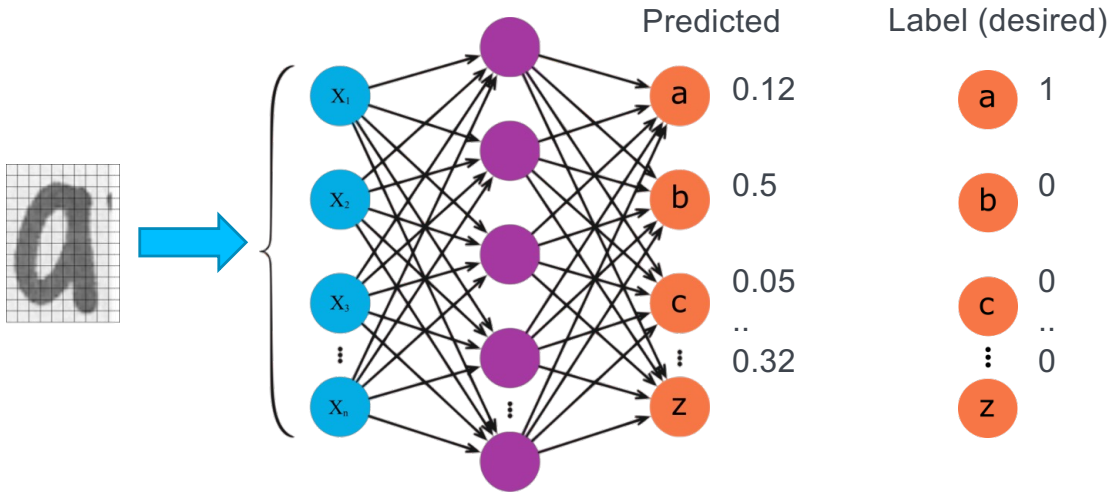


How to build MLP?

- Given the architecture below, where all neurons employ the same activation function (sigmoid)
- We assume that N training samples $(\boldsymbol{x}_i, y_i), i = 1, \ldots, N$ are available.



Neural Networks



Loss function

Predicting probability distributions

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	GAN, VAE, FVBN	Various

6 Modeling a Feed-forward network predicting a Bernoulli distribution

Loss function with maximum likelihood

Given training data $\{(x_i, y_i)\}_{i=1}^{N}, x_i \in \mathbb{R}^d, y_i \in \{0,1\}$

Cross entropy

(as very often used for classification!)

$$L(\theta) = \mathbb{E}_{data}(-\log P(y|\boldsymbol{x})) = -\frac{1}{N}\sum_{i=1}^{N} \log P(y_i|x_i)$$

Bernoulli distribution

Given training data $\{(x_i, y_i)\}_{i=1}^{N}, x_i \in \mathbb{R}^d, y_i \in \{0,1\}$

Model $z = w^T x + b$, **model $\log \boldsymbol{P}(\boldsymbol{y}|\boldsymbol{x}) \sim \boldsymbol{yz}$**

Using the sigmoid activation function, we get

$$P(y|x) = \frac{e^{yz}}{\sum_{y'=0}^{1} e^{y'z}} =$$

} Only large, if $y = 1$ and z is large

} Normalization to 1

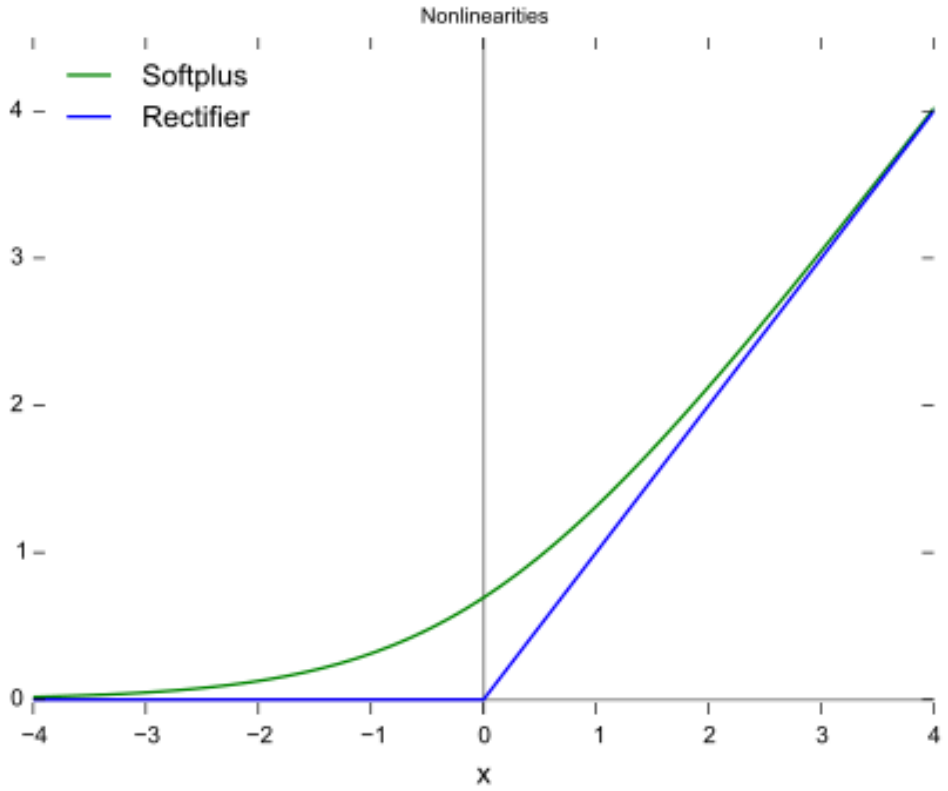
$$= \frac{e^{(2y-1)z}}{1 + e^{(2y-1)z}} = \sigma((2y-1)z)$$

Loss

$$L(\theta) = -\log P(y|x) = -\log \sigma((2y-1)z) = \zeta((1-2y)z)$$

Softplus function ζ

$$\zeta(x) = \log(1 + e^x)$$



CC0, https://en.wikipedia.org/w/index.php?curid=48817276

$$\frac{d\zeta(x)}{dx} = \sigma(x)$$

$$\log \sigma(x) = -\zeta(-x)$$

We consider $\zeta((1-2y)z)$:

- When does ζ saturate?
  - i.e. not provide guidance to the gradient?
- Only if $(1-2y)z$ very negative, i.e.
  - y=1 and z very positive
  - y=0 and z very negative

  i.e. when prediction correct!

Summary: Learning Bernoulli output

- Model:

$$\hat{f}(x) = \sigma(w^T x)$$

- Loss function: Softplus

$$L(\theta) = \zeta((1 - 2y)z)$$

Saturation when correct result achieved

- Solving with gradient descent;

$$\frac{d\zeta(x)}{dx} = \sigma(x)$$

7 Modeling a Feed-forward network predicting a Multinoulli distribution

Comparing Bernoulli vs Multinoulli

Prediction for Bernoulli:

$$\hat{y} = P(y = 1|\boldsymbol{x})$$

With $\hat{y}$ being the predicted value from our neural network for a given input vector $\boldsymbol{x}$

We require

$$\hat{y} \in [0,1]$$

Prediction for Multinoulli:

$$\hat{y}_i = P(y = i|\boldsymbol{x})$$

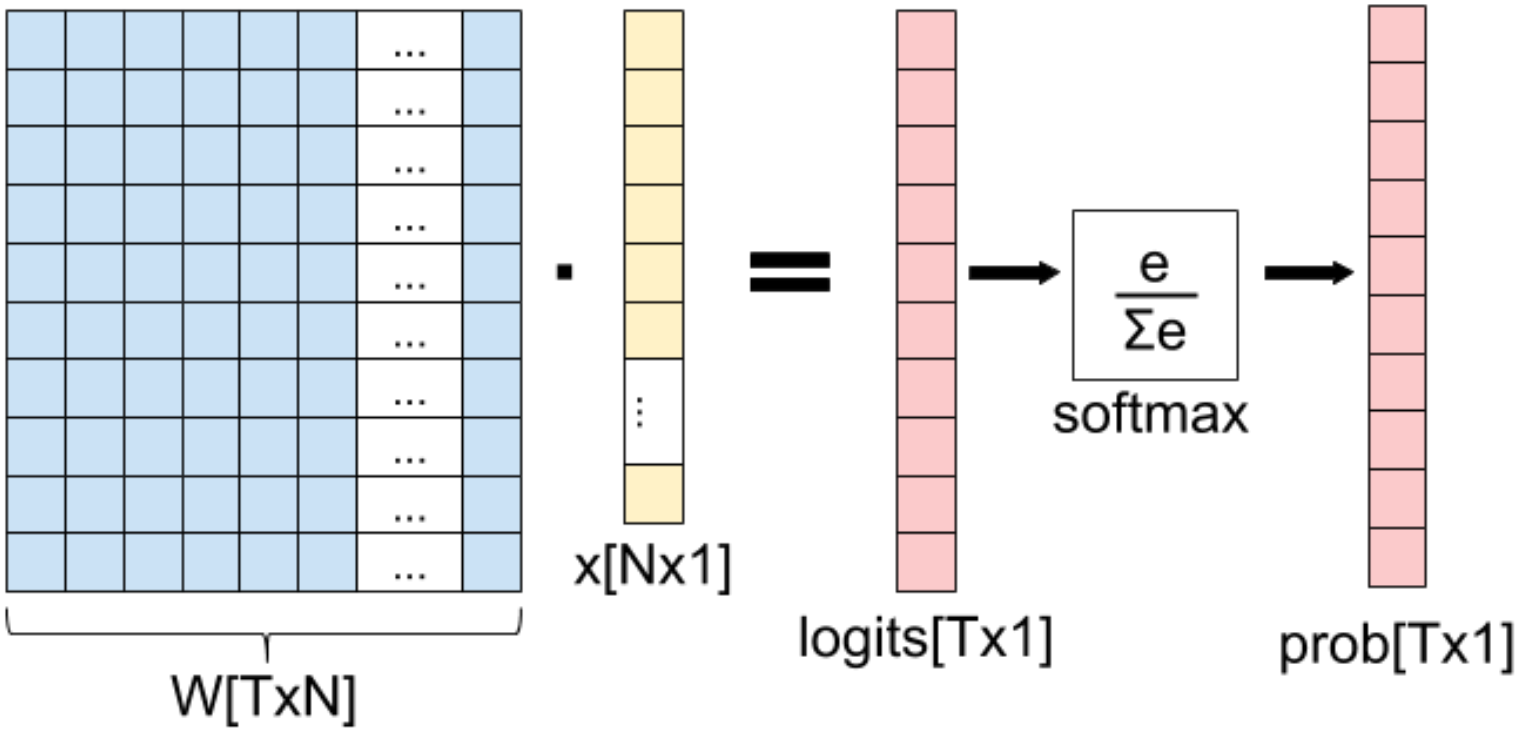
With $\hat{\boldsymbol{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n)$ being the vector of predicted values from our neural network for n different classes

Now we require

$$\forall i: \ \hat{y}_i \in [0,1]$$

$$\sum_{i=1}^{n} \hat{y}_i = 1$$

Multinoulli: Layers



https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/

Bernoulli vs Multinoulli: Layers

Prediction for Bernoulli:

$$\hat{y} = P(y = 1|\boldsymbol{x})$$

We have defined

$$z = \boldsymbol{w}^T \boldsymbol{h} + b$$

scalar vector

Prediction for Multinoulli:

$$\hat{y}_i = P(y = 1|\boldsymbol{x})$$

Now we define

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$

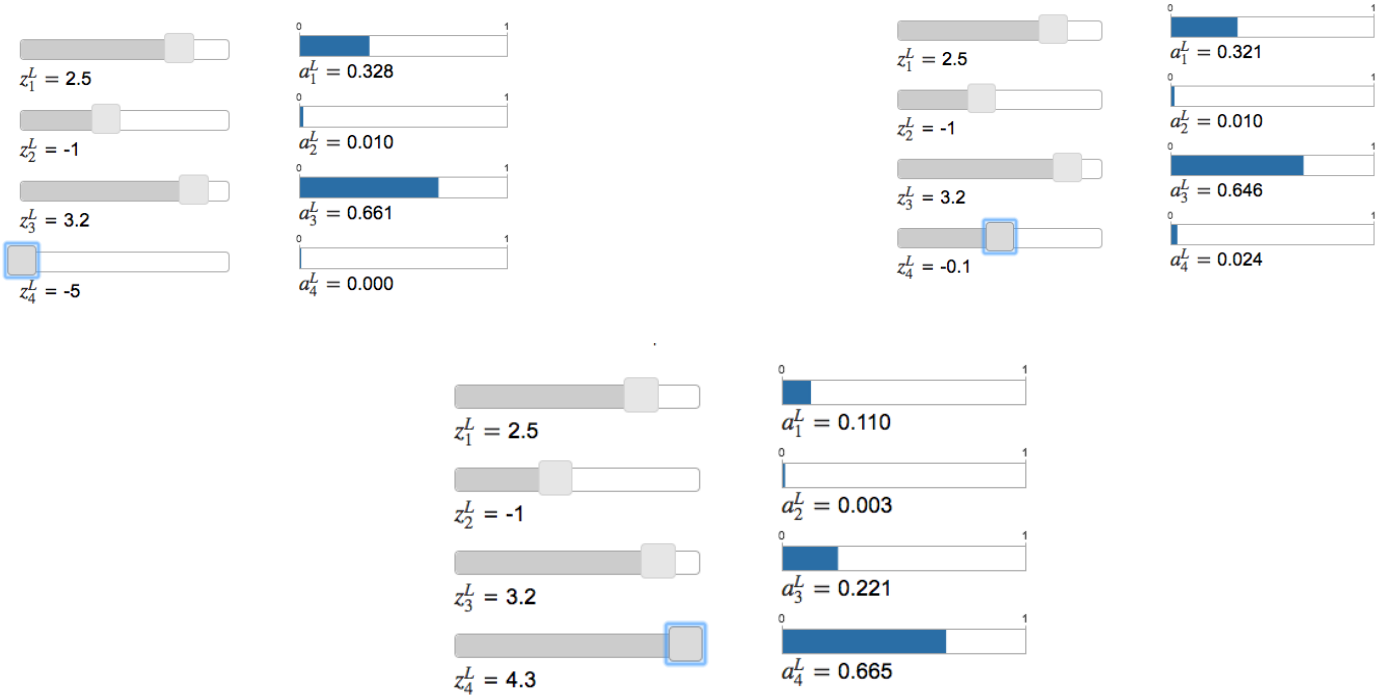
vector matrix

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

scalar

Understanding the softmax

Animation at:
http://neuralnetworksanddeeplearning.com/chap3.html#softmax



Multinoulli: Loss function

Cross entropy for a data record with $y = (0 \ \ldots 0_{\mathrm{i}-1} \ 1_i \ 0_{\mathrm{i}+1} \ \ldots 0)$
(often written in simplified manner as $y = \mathrm{i}$)

data distribution

model distribution learned so far

$$-\sum_j 1_{j=i} \cdot \log \operatorname{softmax}(\boldsymbol{z})_j =$$

$$= -\log \operatorname{softmax}(\mathbf{z})_{\mathrm{i}} =$$

$$= -\log \frac{e^{z_i}}{\sum_j e^{z_j}} =$$

$$= -(\log e^{z_i} - \log \sum_j e^{z_j}) =$$

$$= -(z_i - \log \sum_j e^{z_j})$$

Stable softmax

```
In [146]: softmax([1, 2, 3])
Out[146]: array([ 0.09003057,  0.24472847,  0.66524096])
```

```
In [148]: softmax([1000, 2000, 3000])
Out[148]: array([ nan,  nan,  nan])
```

Remedy:

$$-\log\frac{e^{z_i}}{\sum_j e^{z_j}} = -\log\frac{e^{-c}e^{z_i}}{e^{-c}\sum_j e^{z_j}} = -\log\frac{e^{z_i-c}}{\sum_j e^{z_j-c}}$$

Pick:

$$c = \max_i z_i$$

Summary on Multinoulli

- Softmax as soft generalization of argmax
- „*Winner-takes-most*“

Side note

I liked the explanation by

https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/

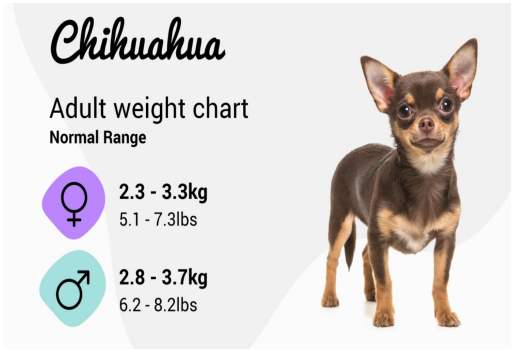
Better than the one in the neural network book

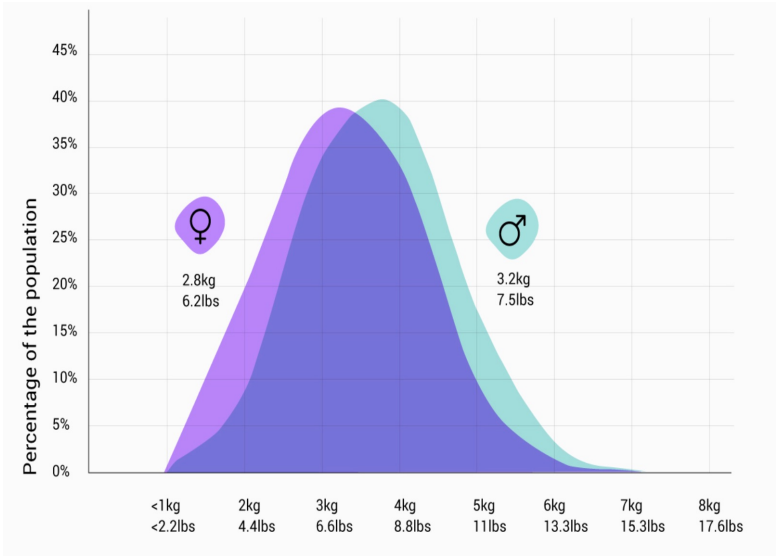
Combining linear output with multinoulli

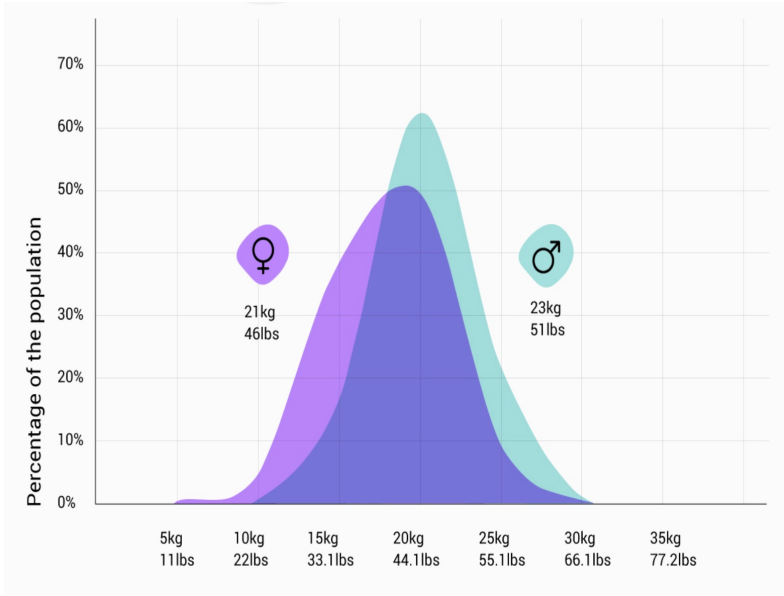
8 Modeling a Mixture Density Network

- (Bishop et al), p. 272
- (Goodfellow et al), chap. 6

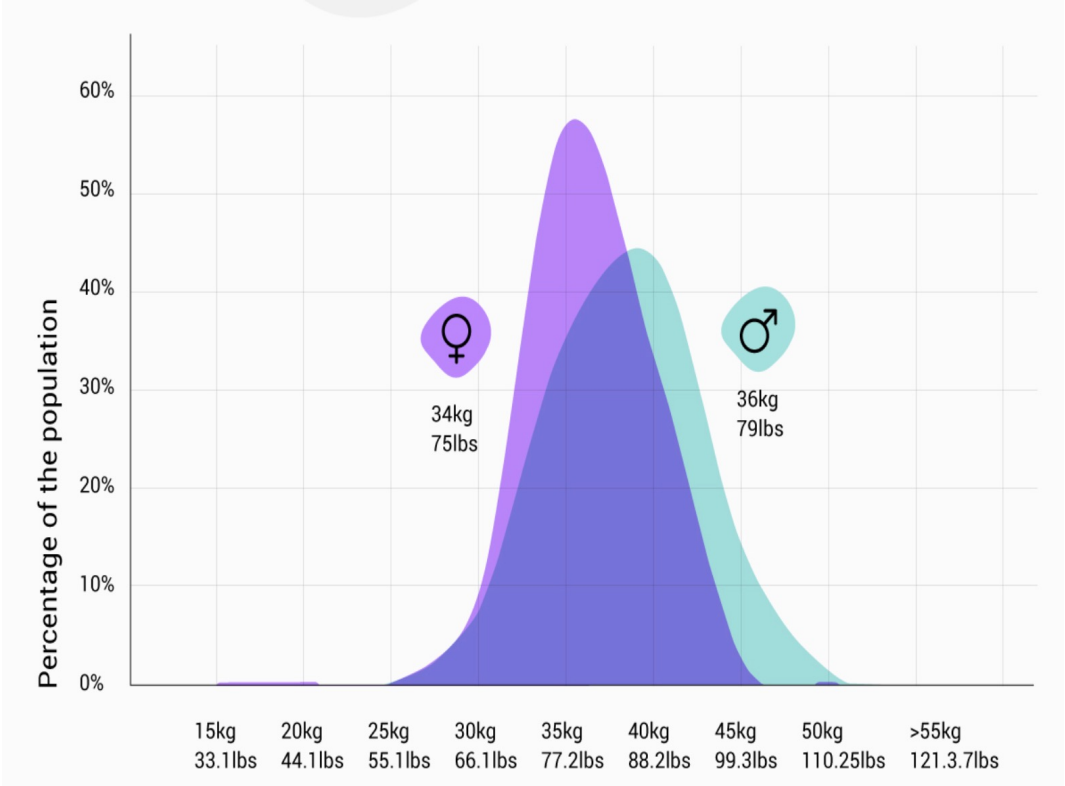
https://www.11pets.com/en/news/siberian-husky-weight-chart







https://www.11pets.com/en/news/german-shepherd-weight-chart



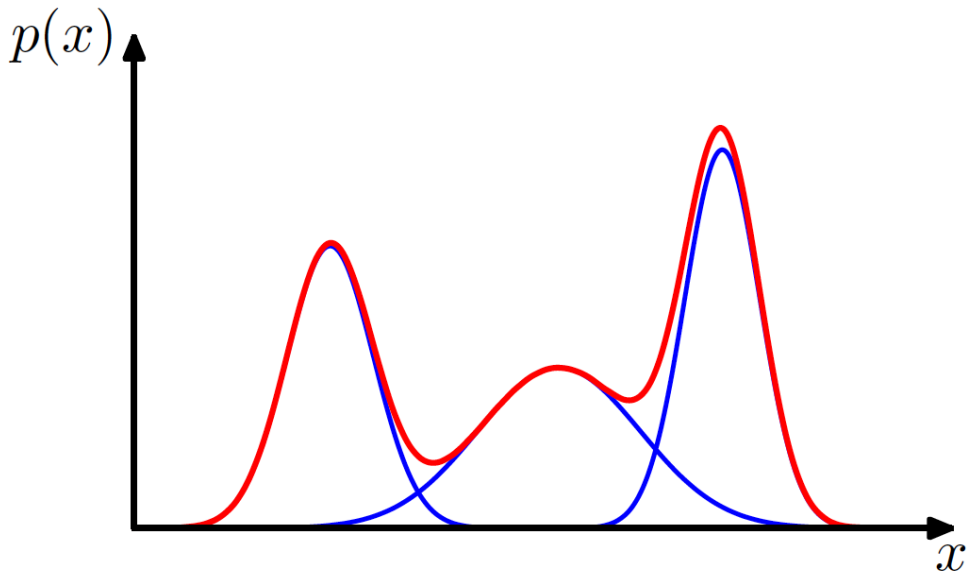
Example

What do we want to achieve? Given an x we want to predict the most probable y.
Let's assume we have four dog breeds: from the tine Chihuahua, over Beagle, and Huskie, to the very large St. Bernard. Let's assume we have a set of dogs and we know some attributes of each dog like TypeOfFood (x1), Location (x2), age (x3), sex (x4) etc., and in the training set in particular we know their weight (y), but we do *not know* their breed. Let's then predict the weight of a previously unseen dog, whose breed we do not know, but whose other attributes (x vector) we know.
We might do a linear regression to predict the weight which will vary from 1.8–2.7 kg (for the Chihuaha) to 54-82kg for the St. Bernard. This prediction will utterly fail, because for each of the four breeds we will have a very different mean of weight values.
Hence, what we want to do is to consider the breed without knowing the breed. We want to discover in the data that there are four different clusters (around 2kg for the Chihuahua, 10kg for the Beagle, 20kg for the Husky, 70kg for the St. Bernard) and given the data (e.g. location and food) we might want to find that attributes like location tell us something about which breed is more probable than another breed and consider this for the prediction of the weight, where the weight for each cluster is approximately normal distributed around the center with larger variations along some axes (e.g. sex) and less variation along other axes.
The weight of each breed is approximately Gaussian distributed. Hence, trying to do both at the same time, i.e. finding the implicit cluster, its mean and its variation along the axes (e.g. St. Bernard's weight varies in the range of 64-82kg for males and 54 to 64kg for females, this is way more variation than one could find for the Chihuahua) allows us to do a joint prediction given some other attributes of the most probable weight value (y).

Actual usage example would be in the area of prediction of vehicle movements, but there the assumptions are much more difficult than in the (unrealistic) dog example.

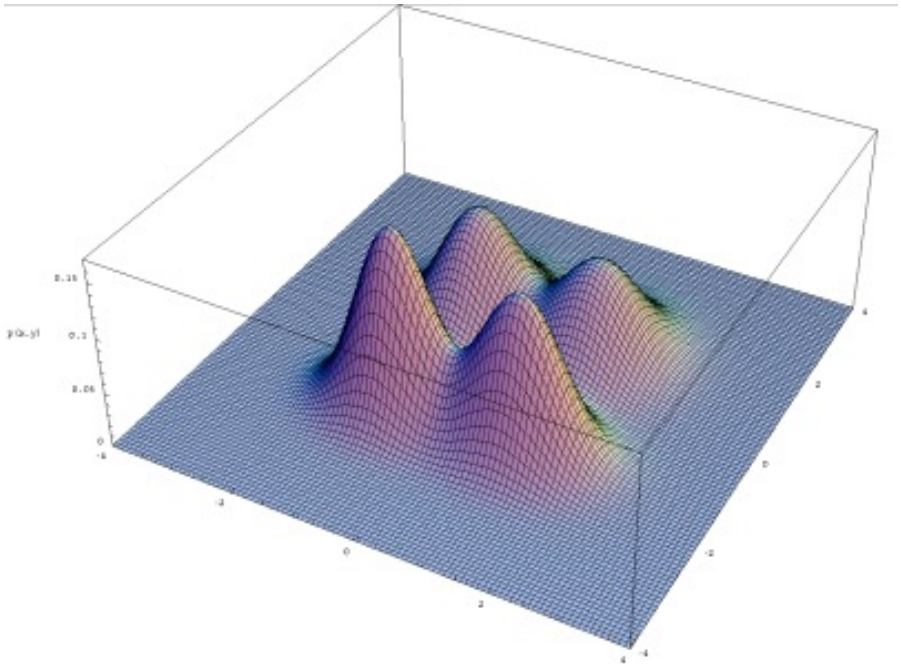
Regression of a Gaussian Mixture

Univariate mixture distribution, showing bimodal distribution



(Bishop et al.), page 111

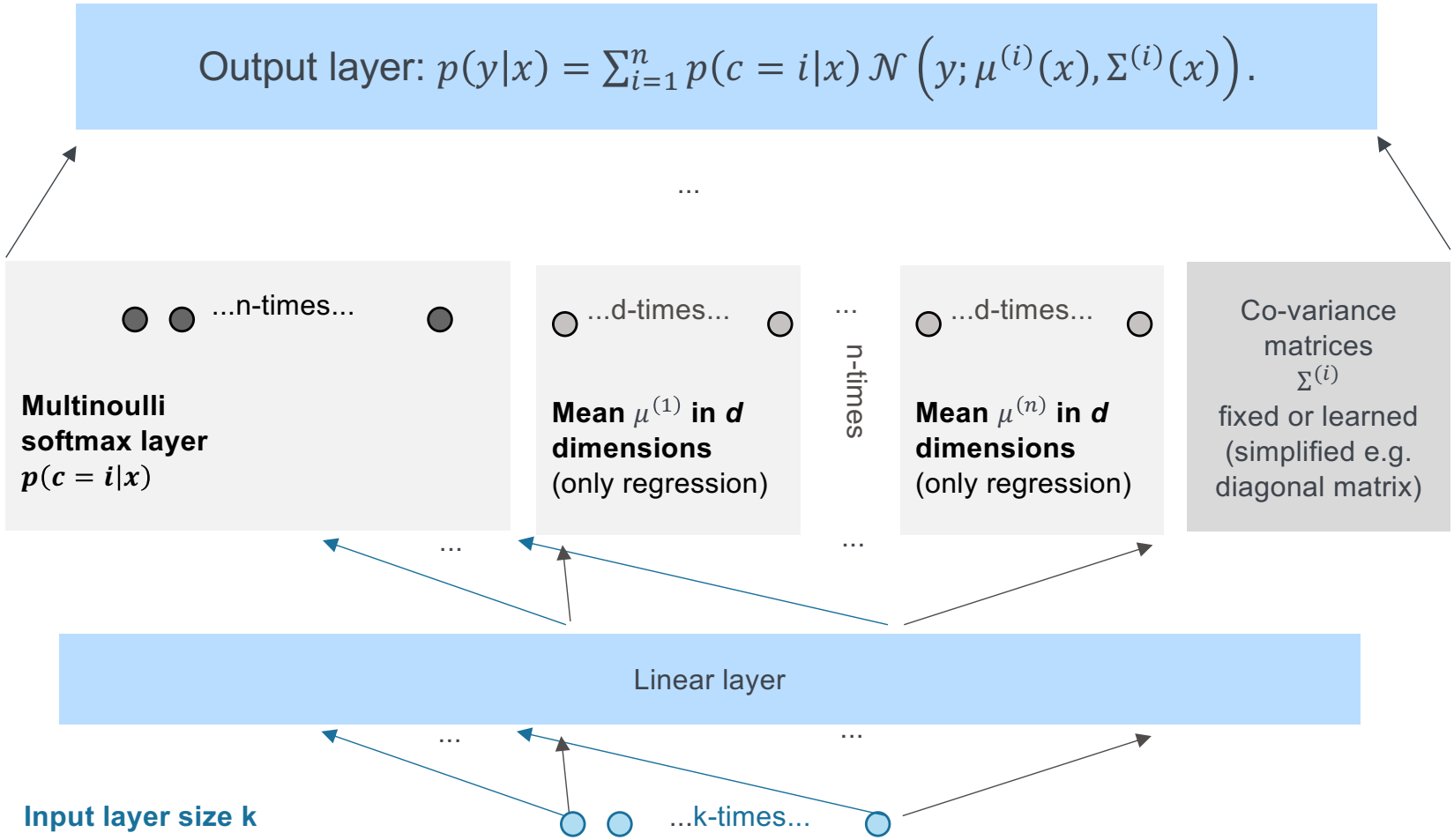
Regression of a Gaussian Mixture
Multivariate (bi-variate) mixture distribution, showing four modes



By Jorgenumata (talk) (Uploads) (Own work) [Public domain], via Wikimedia Commons

https://upload.wikimedia.org/wikipedia/commons/1/12/Bimodal-bivariate-small.png

Regression of aGaussian mixture



Predicting

$$\mathbb{E}[y|x] = \int y\, P(y|x)dy = \sum_{i=1}^{n} P(c = i)\mu^{(i)}(x)$$

Gaussian mixture

- Different strategies for co-variance matrix
  - Constant
  - Diagonals (with partial influence according to category)
  - Full (with partial influence according to category)
- Loss function:
  - Mean Squared Error
    - Influences the means
    - Influences the choice of categories
      - make the dependence continuous
    - A mean should only be corrected if its category was influential

Watch out in winter term 2025/2026!

Seminar

Machine Learning on Graphs

by

Hernandez and team

Fachpraktikum

Deep Learning Lab

Nayyeri et al

Watch out in winter term 2025+26!

Lecture + Project

Foundation Models / Deep Generative Models

by

Mathias Niepert, Steffen Staab, Kai Arras

from BERT to ChatGPT, DALL-E, Llama etc.

Limited participation

Universität Stuttgart
KI

Thank you!

Steffen Staab

E-Mail Steffen.staab@ki.uni-stuttgart.de
Telefon +49 (0) 711 685-88100
www. ki.uni-stuttgart.de/

Universität Stuttgart
Analytic Computing, KI
Universitätsstraße 32, 50569 Stuttgart