Deep Feedforward Networks

(Feedforward Neural Networks)
(Multi-layer Perceptrons, MLPs)

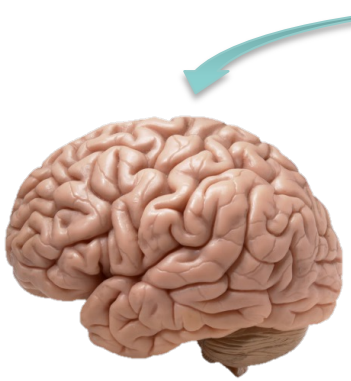
Yong-Sheng Chen
Dept. Computer Science, NYCU

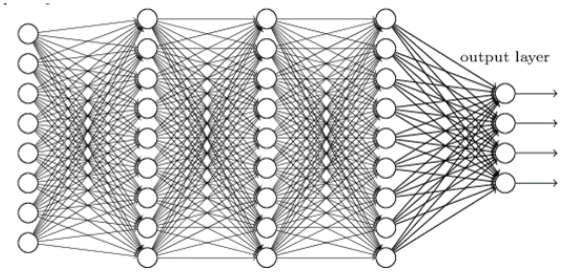
Sources:
"Neuroscience---exploring the brain," 2nd ed.
by Bear, Connors, and Paradiso
Ch. 6, "Deep Learning" textbook by Goodfellow et al.
Slides by Prof. Wen-Hsiao Peng

Human neural network vs. artificial neural network

Technology for functional brain mapping

- Functional brain mapping
- Structural analysis





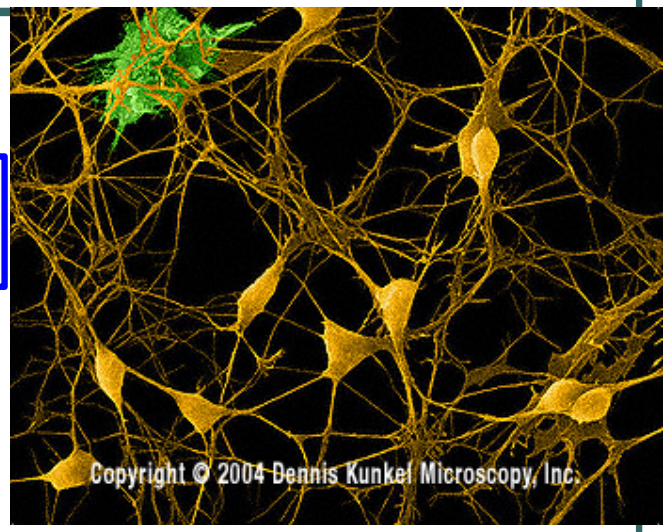
- Computer vision
- CAD
- etc.

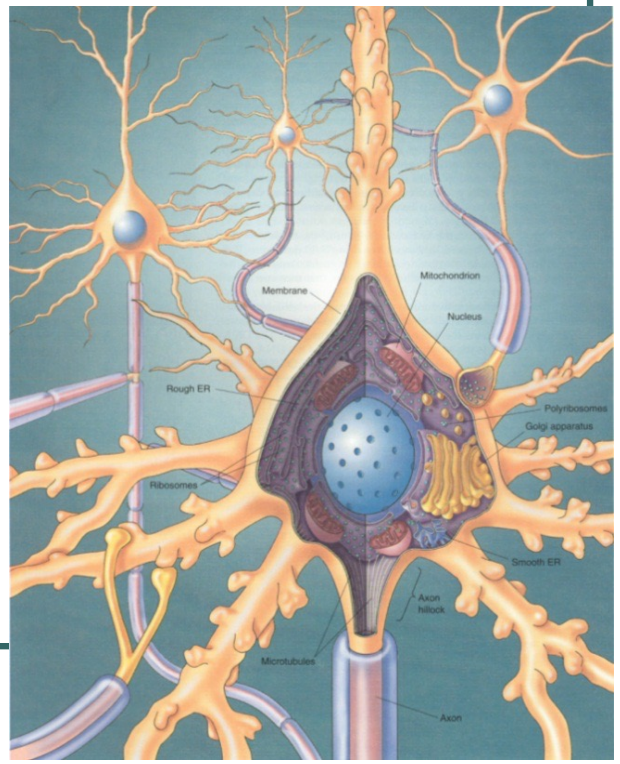
Bio-inspired deep neural network architecture

Cortical Neurons

- 10% brain cells are neurons. 90% are glial cells
- ~ 100 billion (100 G) neurons
- ~ 2500 cm^2 of 2~4 mm thick sheet
- We loss >10000 neurons per day.
- 1.5 million km of fibers
- Neurons work collectively.
- 2% of adult body's weight, but 20% of its energy consumption

NVidia A100:
6192 CUDA cores

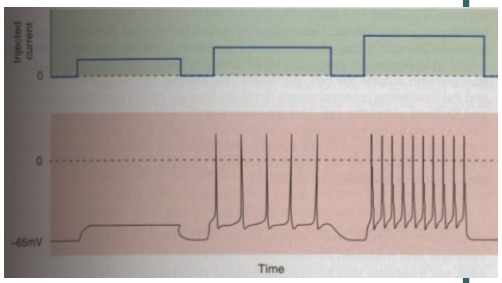


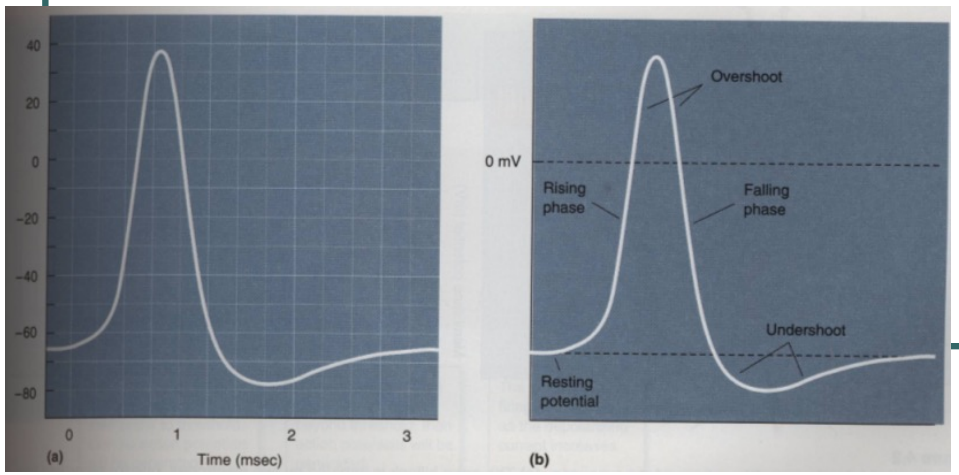


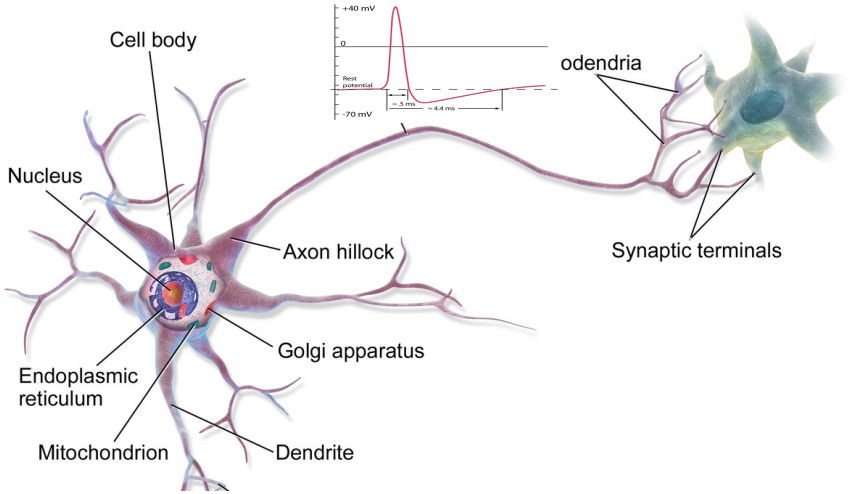
Connors et al. 2002,
Neuroscience: exploring the brain

Action Potential

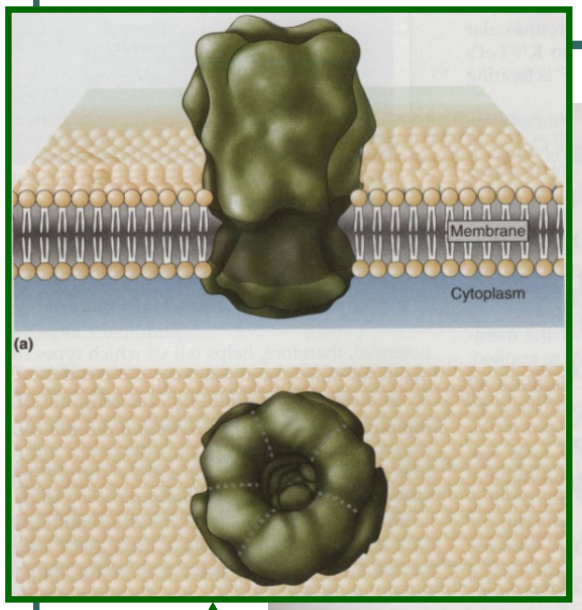
- Action potential
  - triggered by depolarization of the membrane beyond a threshold (40mV for a typical neuron)
  - the basic component of all bioelectrical signals
  - conveys information over distances
  - caused by the flow of sodium (Na^+), potassium (K^+), chloride (Cl^-), and other ions across the cell membrane
  - all-or-none phenomenon
  - frequency and temporal pattern constitute the code

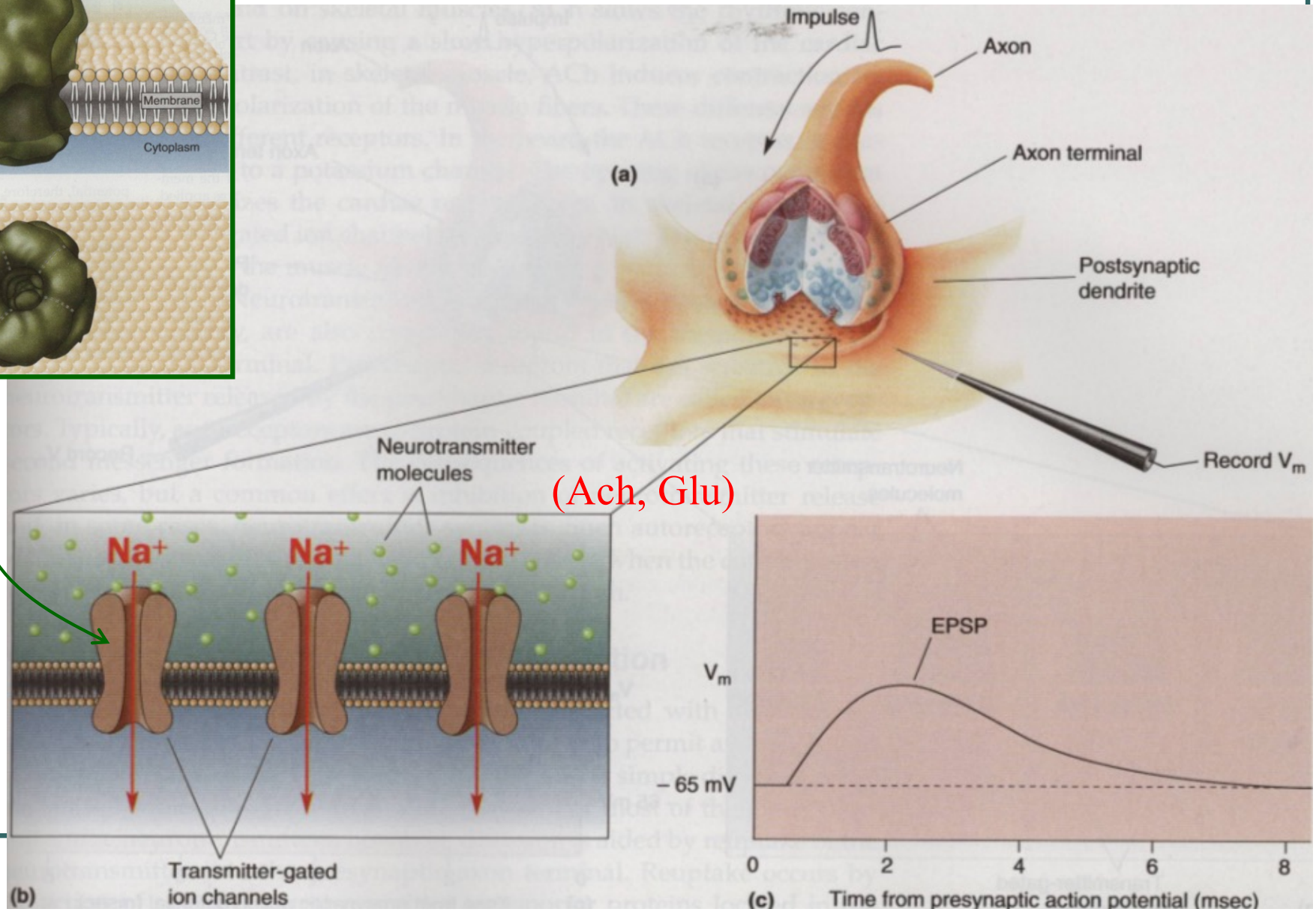




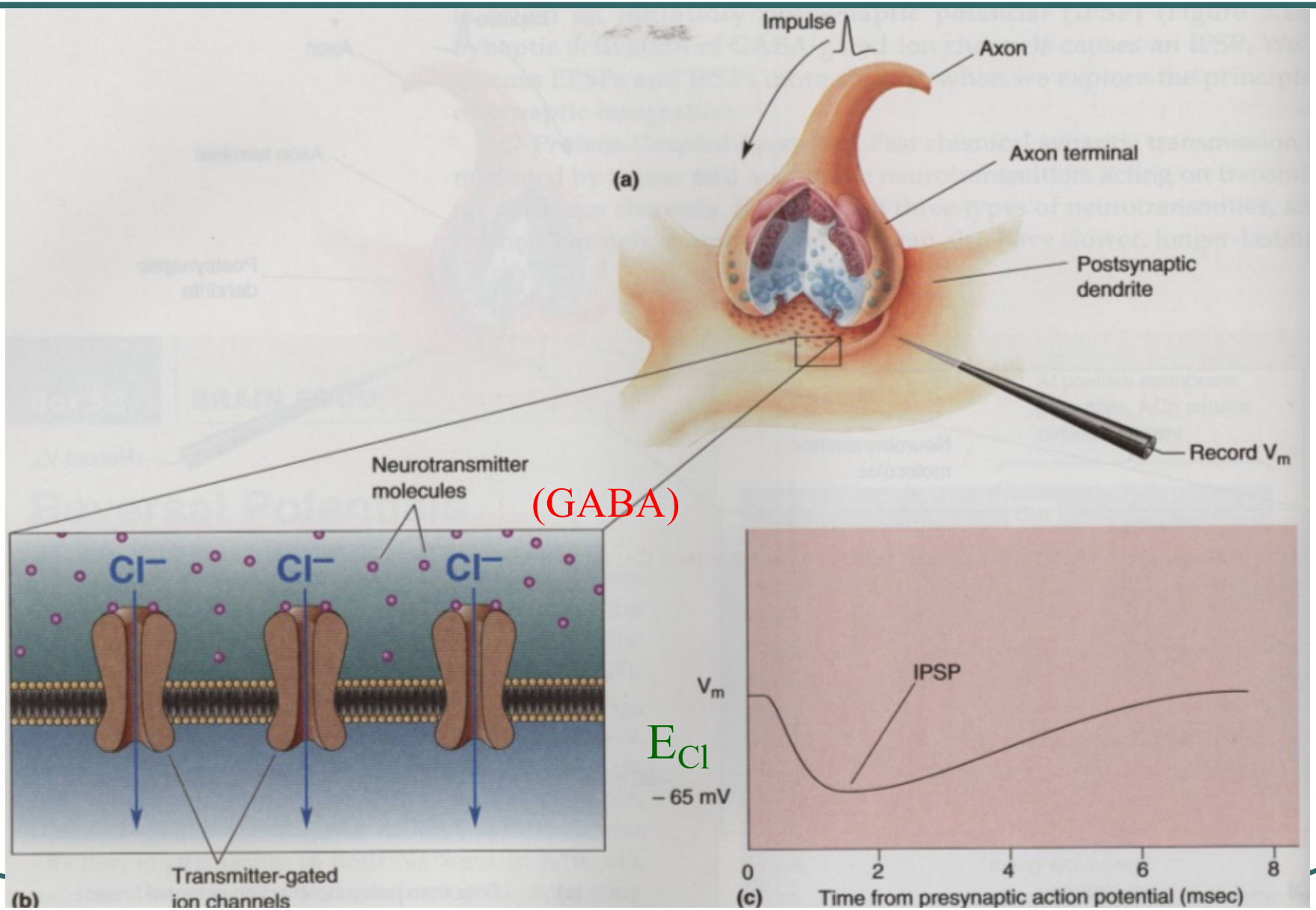


Excitatory Postsynaptic Potential (EPSP)

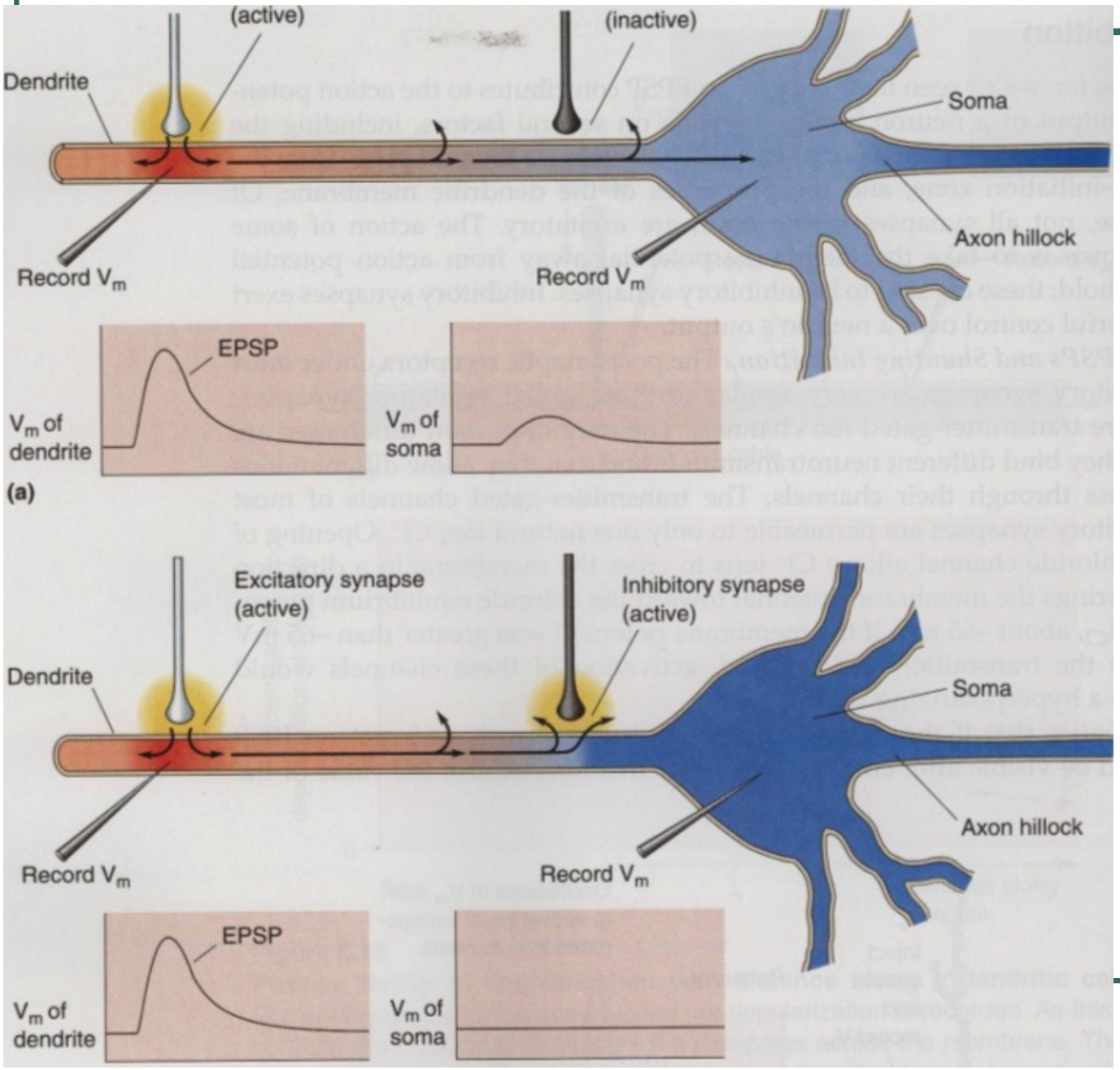




Inhibitory Postsynaptic Potential (IPSP)

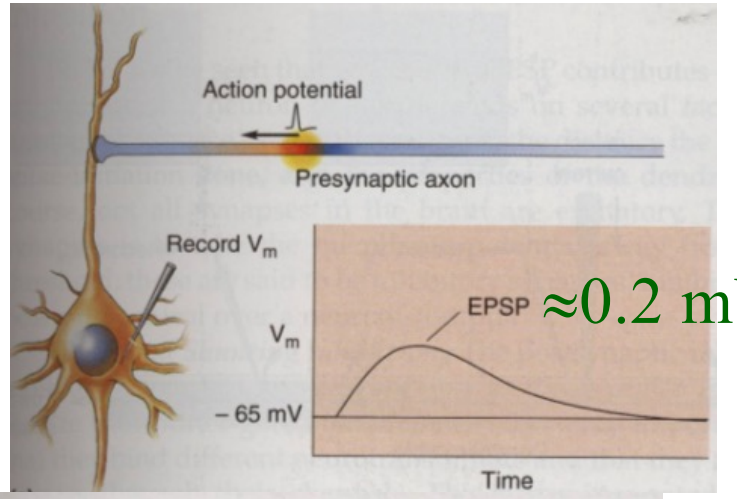


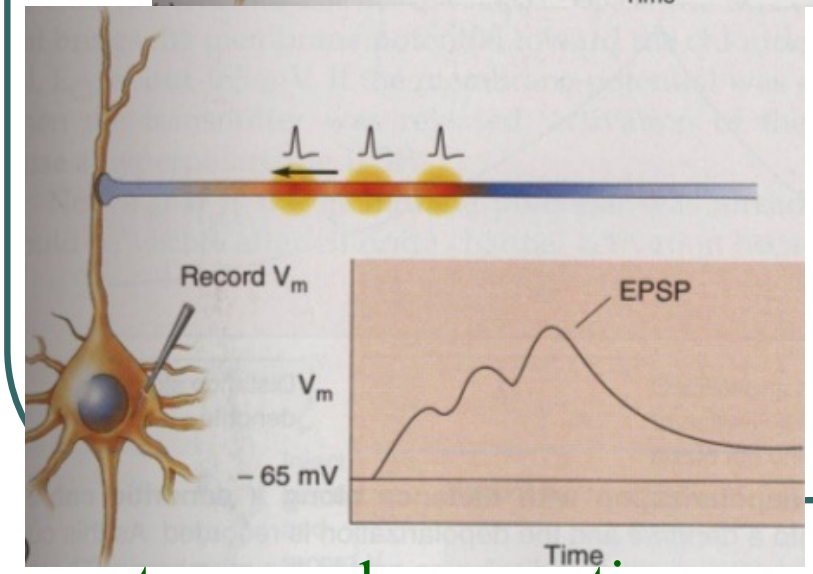
IPSP and Shunting Inhibition



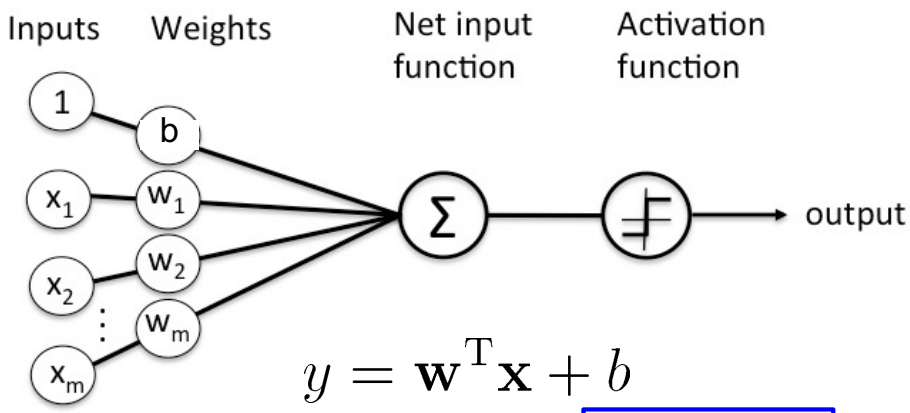
EPSP Integration

- Neurons in CNS perform computations.



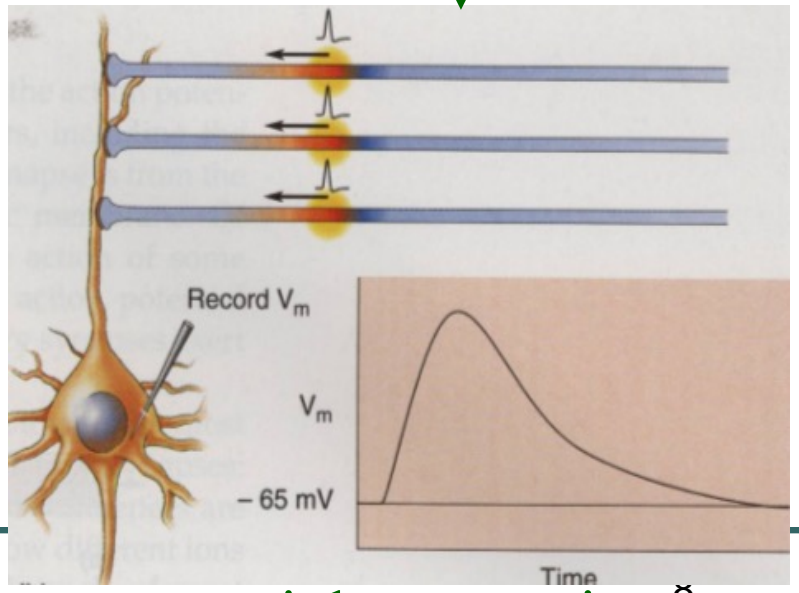


temporal summation



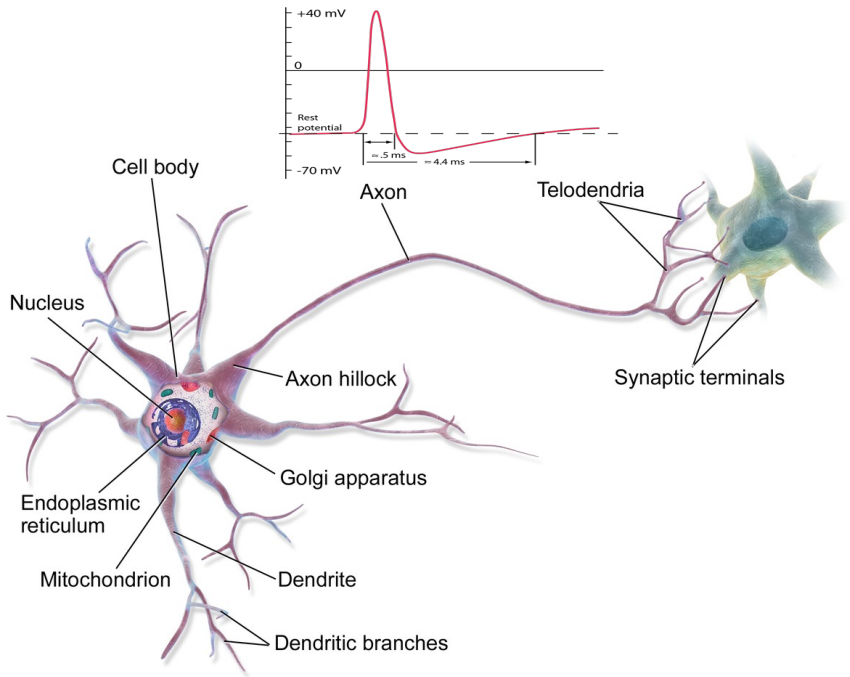
$$y = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b$$

b: -40mV

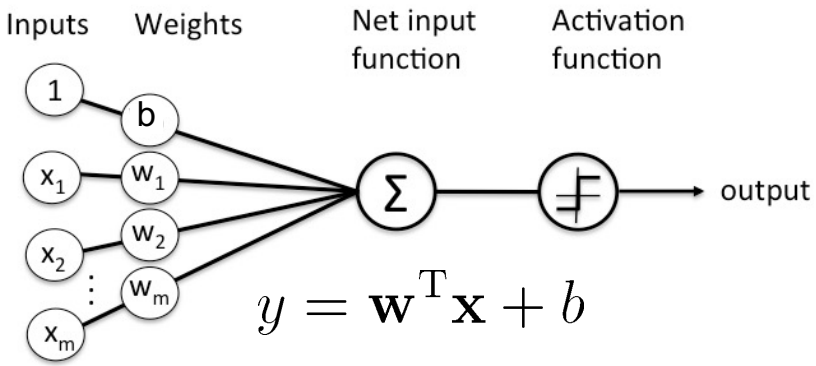


spatial summation

Neuron and Perceptron

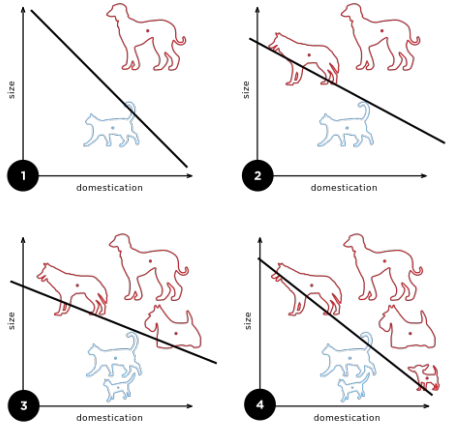


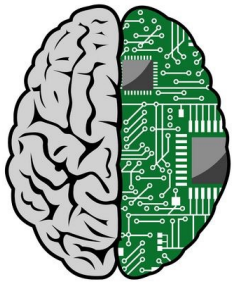
A biological neuron



An artificial neuron (Perceptron)
- a linear classifier

Frank Rosenblatt
1957

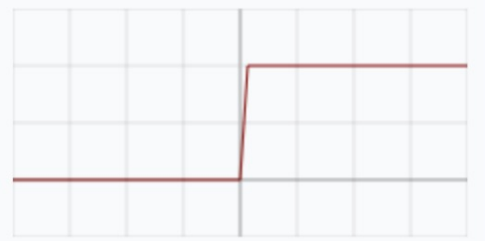




Activation Functions

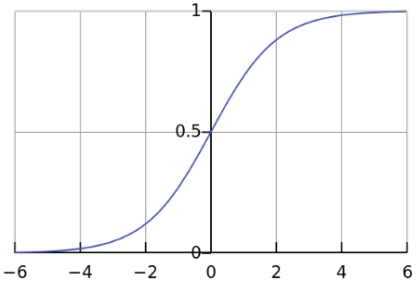
- Nonlinearity of neural network
- Binary step function

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$$



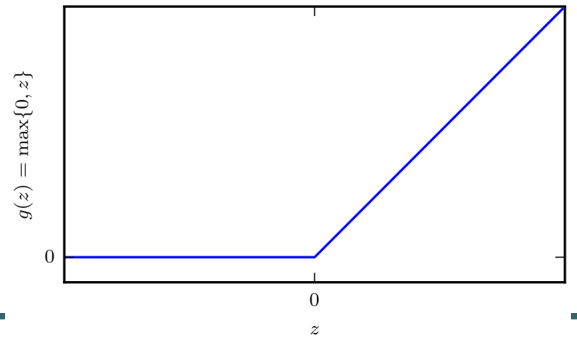
- Sigmoid function

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

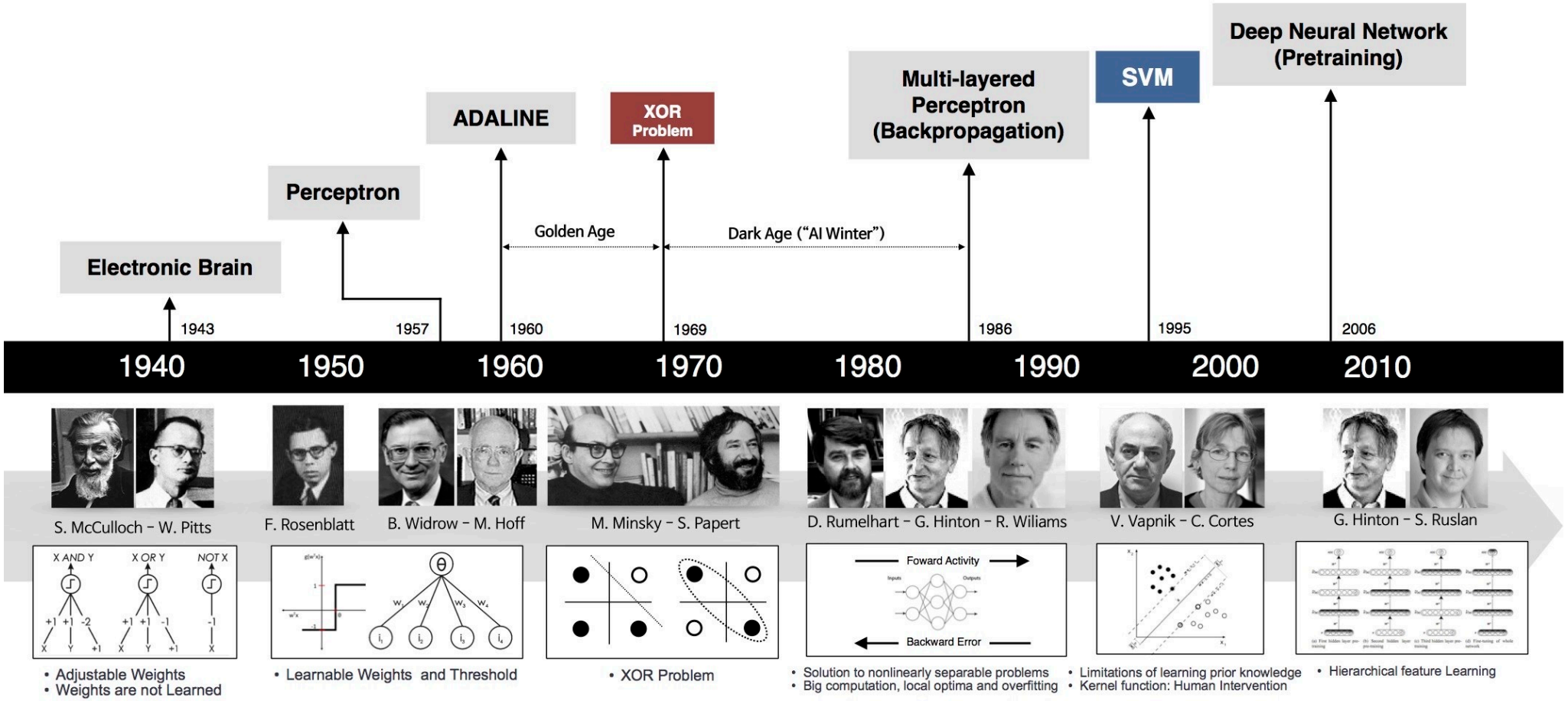


- ReLU: rectified linear unit

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

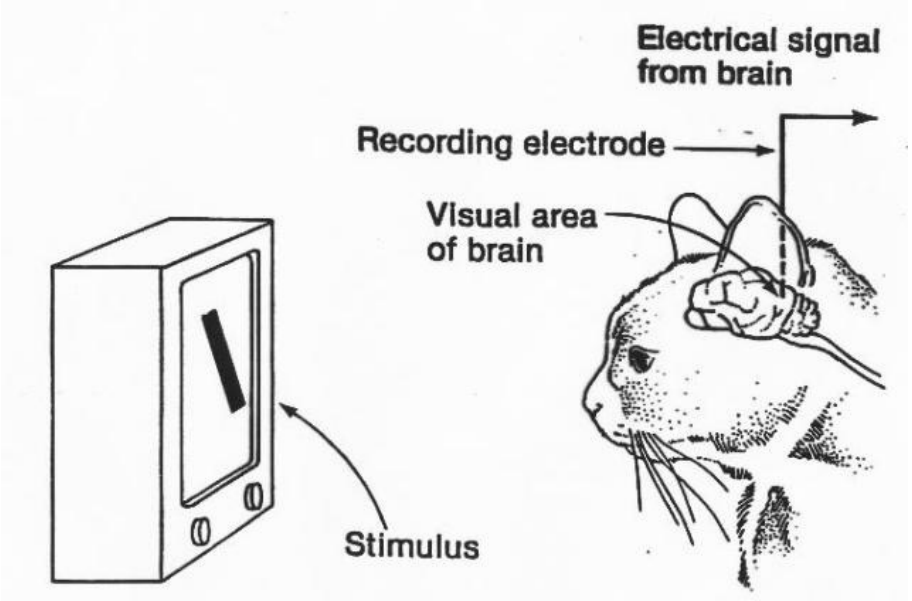


History of Neural Network

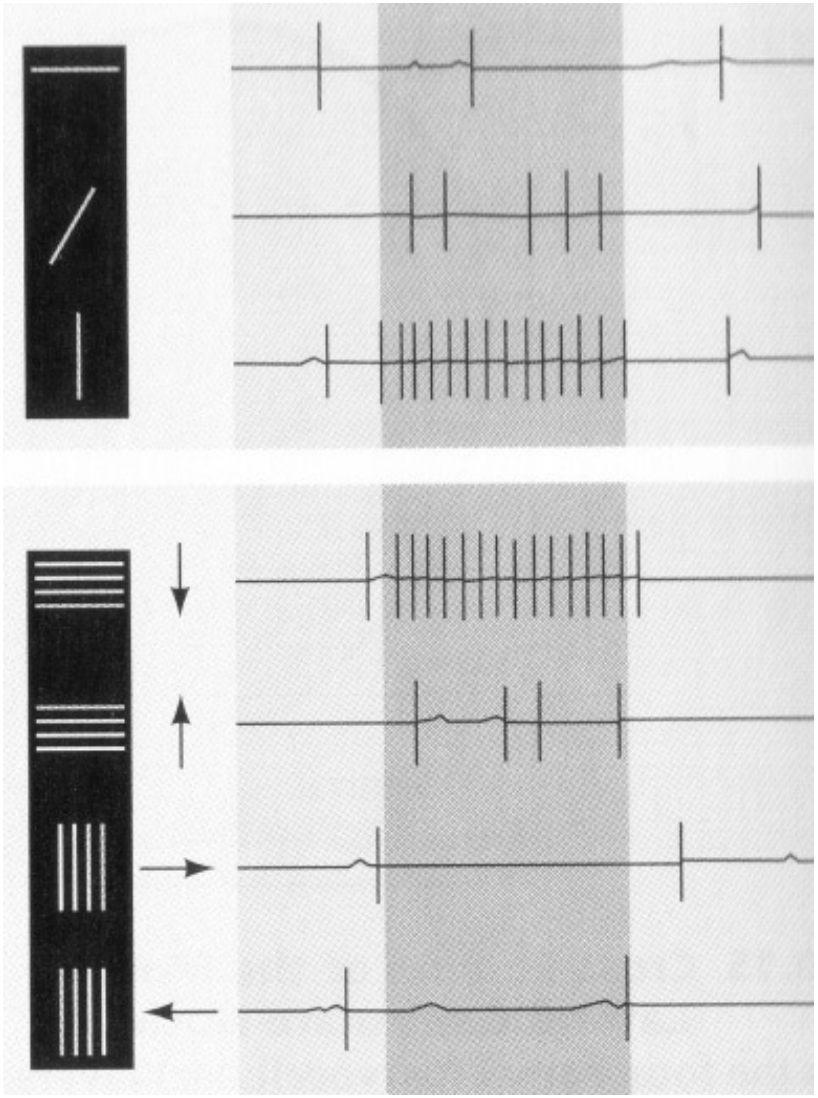


beamlab

Receptive Fields of Lateral Geniculate and Primary Visual Cortex

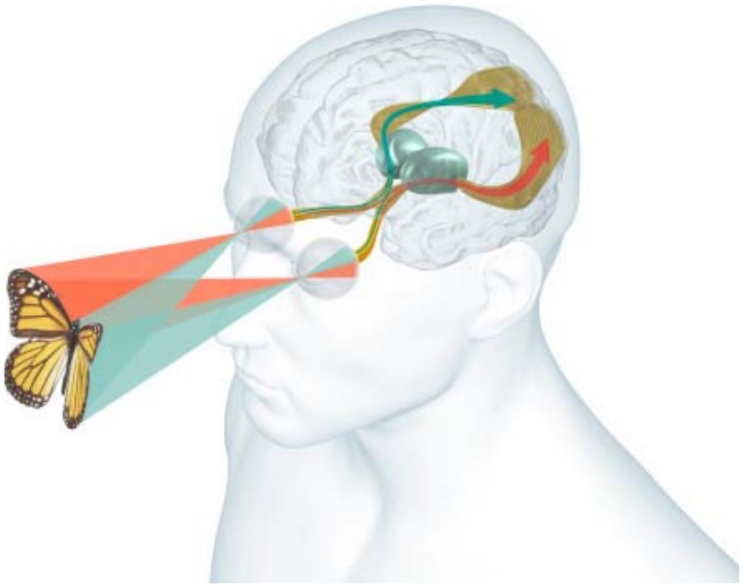


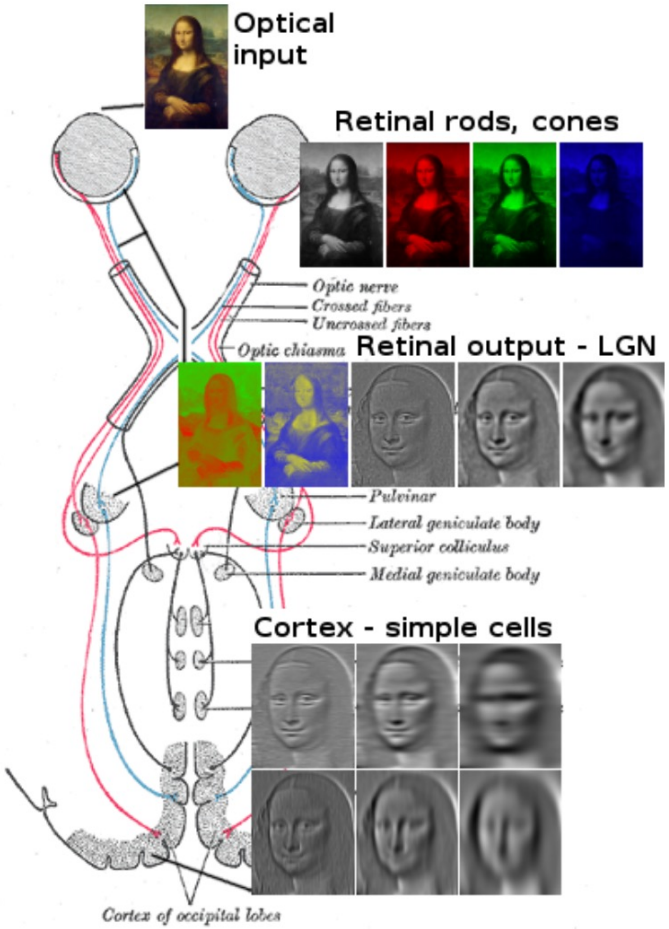
David Hubel and Torsten Wiesel, 1962

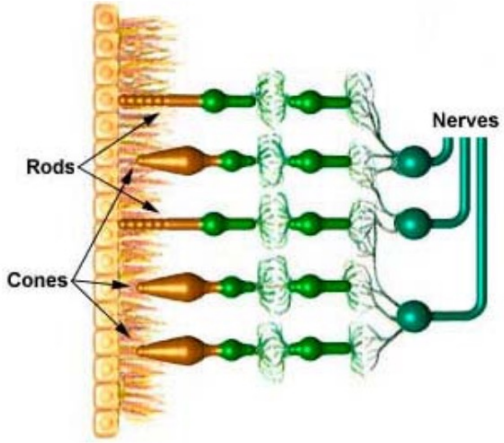


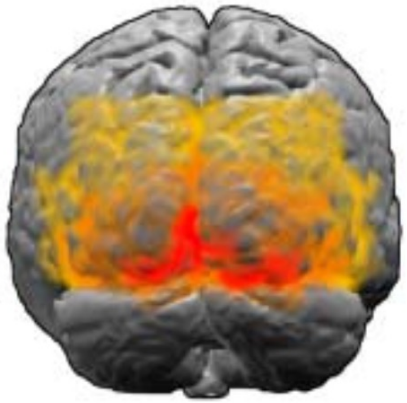
From Components to Systems

Eye/Brain combination

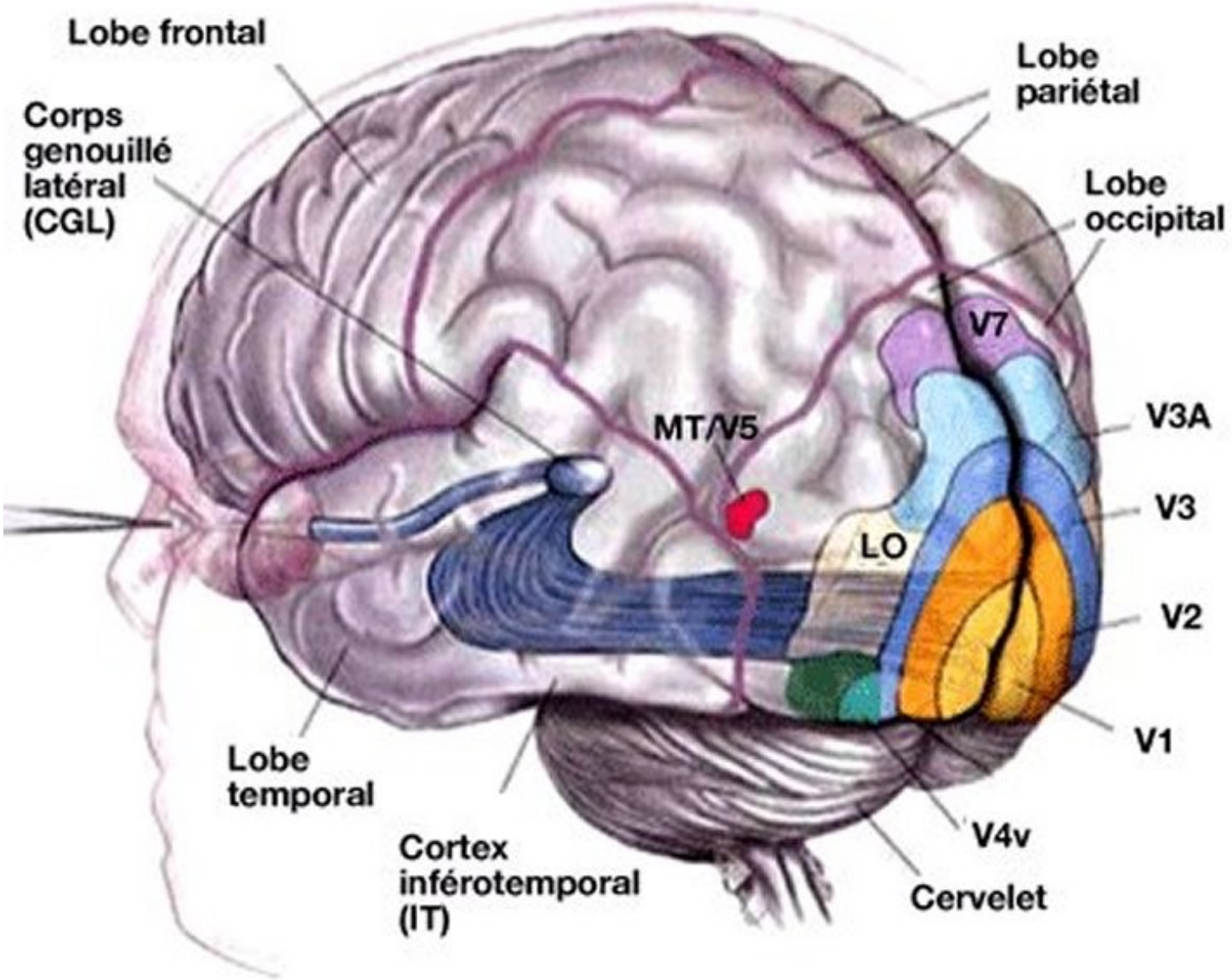








Human Cortical Visual Regions: V1, V2, V3, V4, V5 (MT)

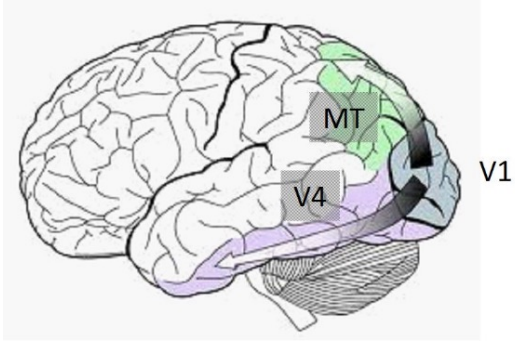


V1: orientation

V2: corner

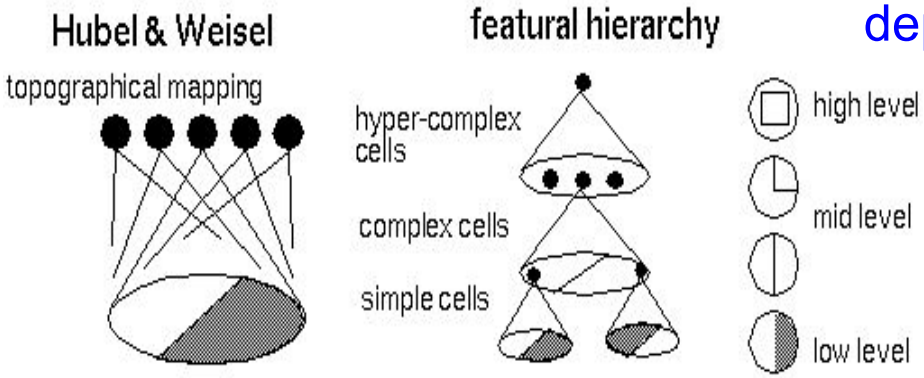
V4: color

IT: object recognition

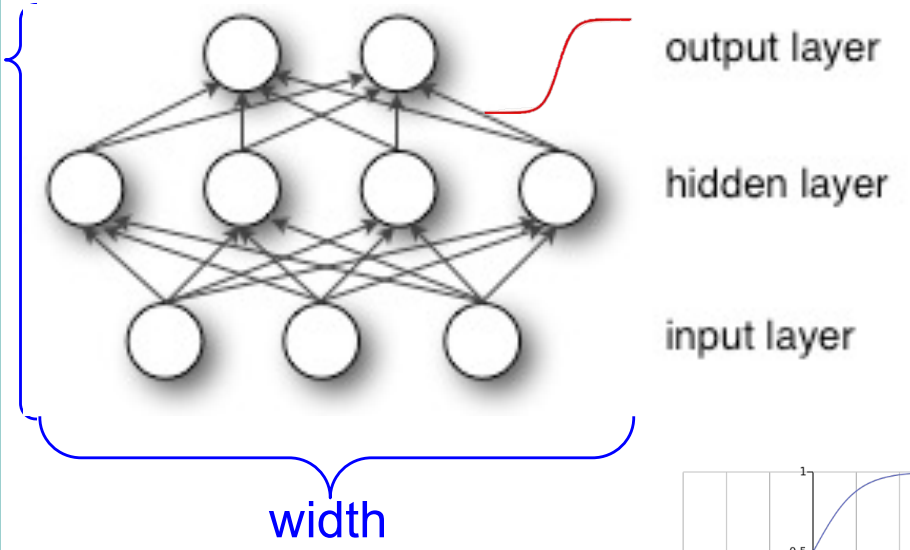


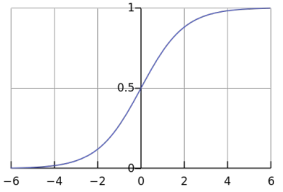
http://raymond.rodriguez1.free.fr/Documents/Organisme-A/Vision

Hubel/Wiesel Architecture and Multi-layer Perceptrons

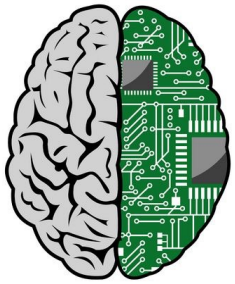


Hubel and Weisel's architecture





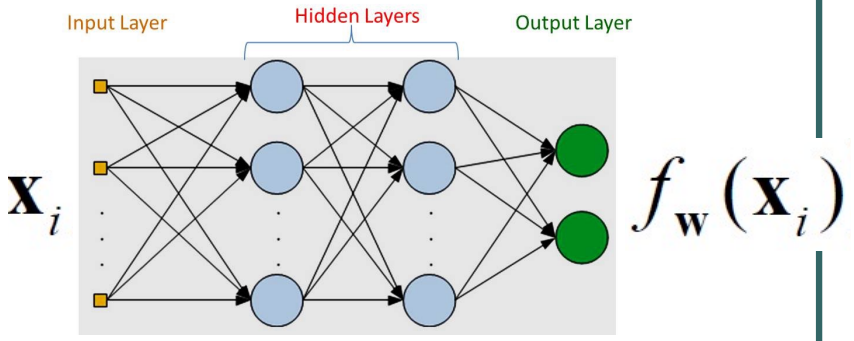
Multi-layer perceptrons
- A *non-linear* classifier with sigmoid activation function



Multi-layer Perceptrons

- A non-linear classifier
- **Training:** find network weights **w** to minimize the error between true training labels y_i and estimated labels $f_w(\boldsymbol{x_i})$

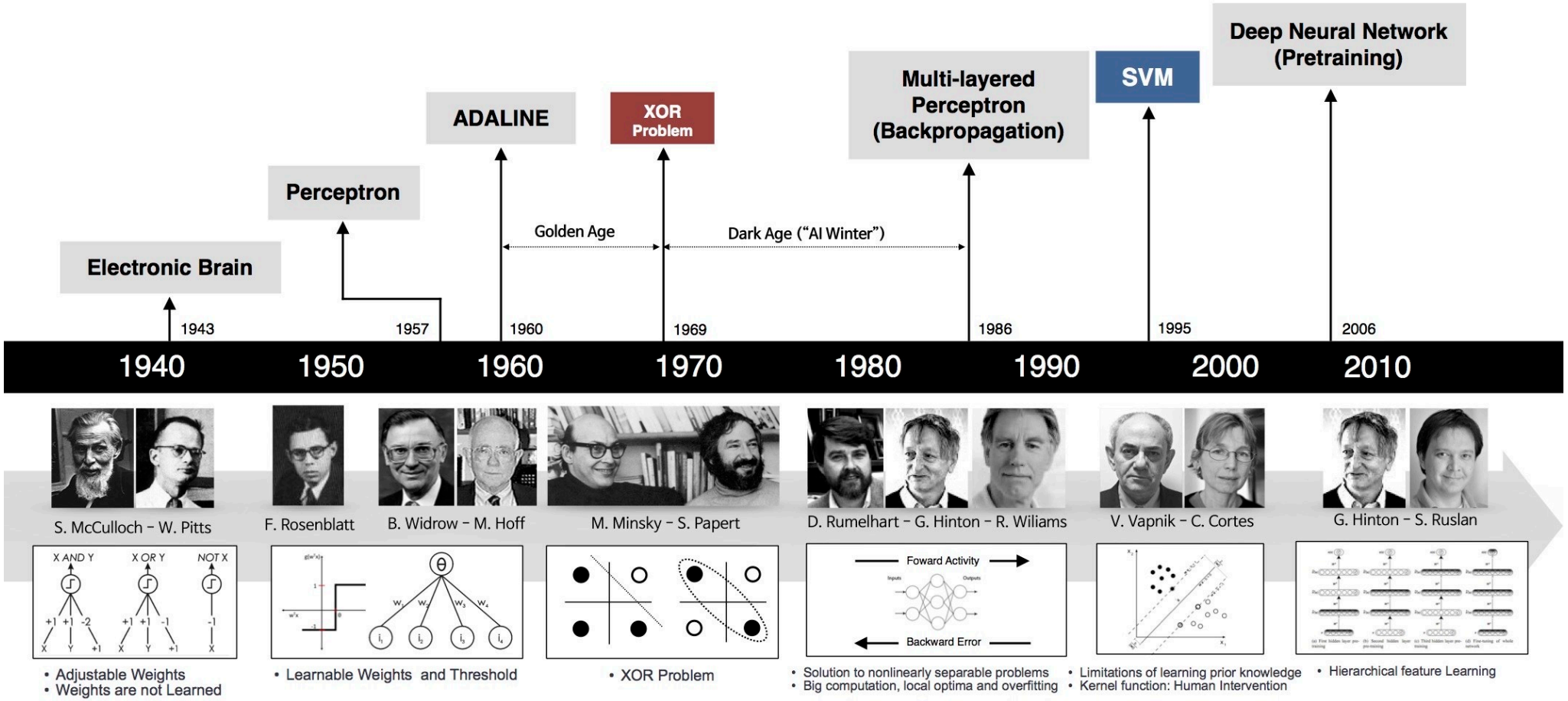
$$E(\mathbf{w}) = \sum_{i=1}^{N} \left(y_i - f_{\mathbf{w}}(\mathbf{x}_i) \right)^2$$



- Minimization can be done by gradient descent provided f is differentiable
- This training method is called **back-propagation**

D. Rumelhart, G.E. Hinton, and R. J. Williams,"Learning Internal Representations by Error Propagation," Nature, 1986.

History of Neural Network



beamlab

From linear to nonlinear

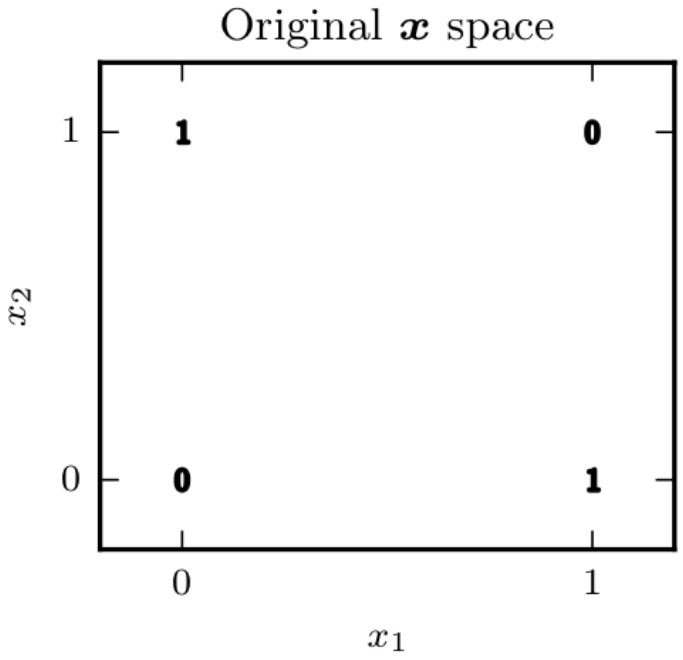
- To extend linear models to represent nonlinear function of ***x***: $y = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b \longrightarrow y = \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}) + b$
  - To use kernel functions such as radial basis functions (RBFs), e.g.: $\phi(\mathbf{x}) = e^{-(\epsilon\|\mathbf{x}-\mathbf{x}_i\|)^2}$
  - Manually engineer ϕ, that is, features in computer vision, speech recognition, etc.
  - To learn ϕ from data:

$$y = f(\mathbf{x}; \theta, \mathbf{w}) = \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}; \theta)$$

A simple example: learning XOR

- Data: $\chi = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$
- Target function: $y = f^*(\chi) = \{0, 1, 1, 0\}$
- Linear model: $y = f(\mathbf{x}; \theta = \{\mathbf{w}, b\}) = \mathbf{x}^{\mathrm{T}}\mathbf{w} + b$
- MSE loss function:

$$J(\theta) = \frac{1}{4} \sum_{\mathbf{x} \in \chi} (f^*(\mathbf{x}) - f(\mathbf{x}; \theta))^2$$



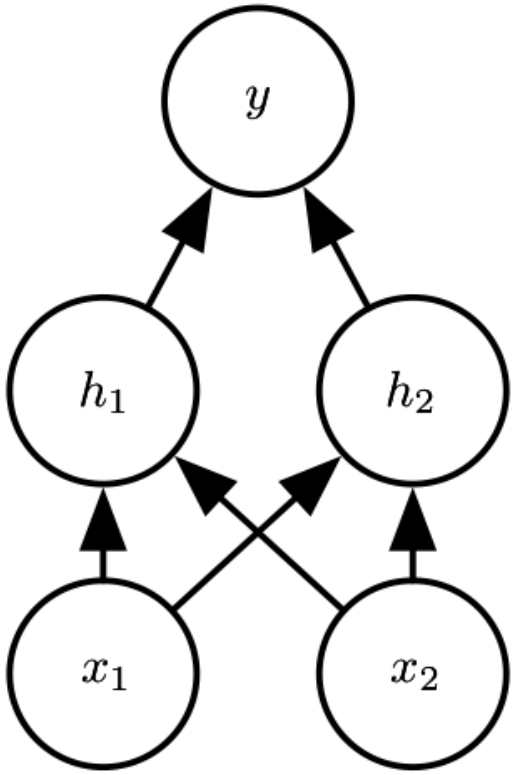
- Linear model is NOT able to represent XOR function.

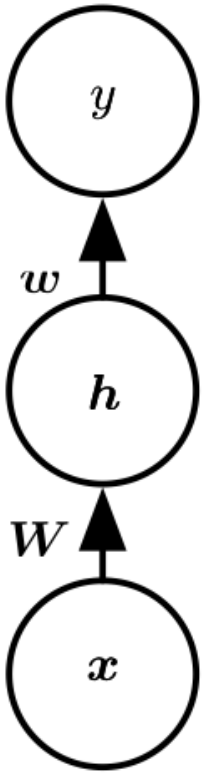
A simple example: learning XOR

- Use one hidden layer containing two hidden units to learn ϕ.

$$y = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b \longrightarrow y = \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}) + b$$

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





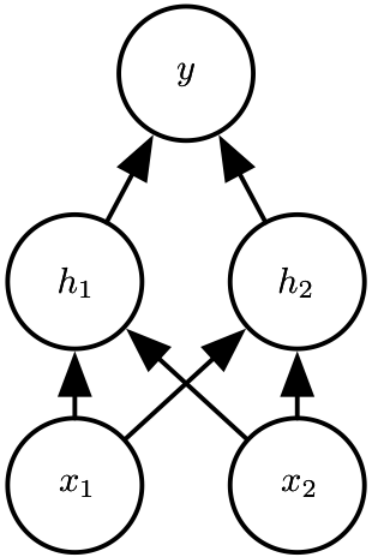
$$y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$$

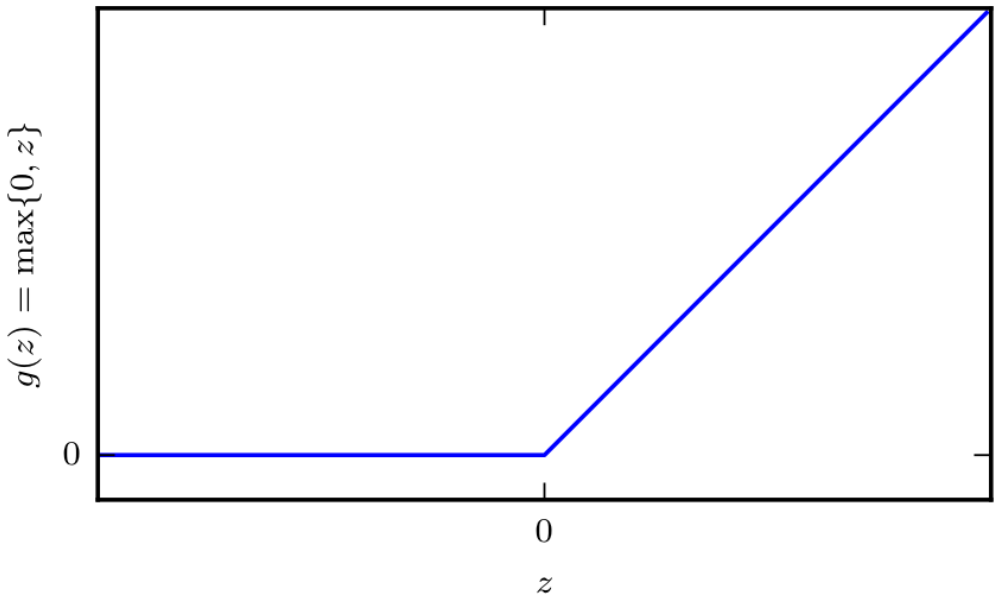
$$\phi(\mathbf{x}) = \mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$$

A simple example: learning XOR

- Let there be nonlinearity!
- ReLU: Rectified Linear Unit: $g(z) = \max\{0, z\}$
- ReLU is applied element-wise to ***h***:

$$h_i = g(\mathbf{x}^{\mathrm{T}}\mathbf{W}_{:,i} + c_i)$$





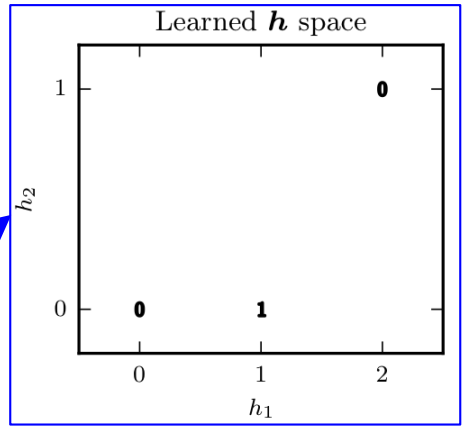
A simple example: learning XOR

- Complete neural network model:

$$y = f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x})) = \mathbf{w}^{\mathrm{T}} \max\{0, \mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{c}\} + b$$

- Obtain model parameters after training:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$



- Run the network:

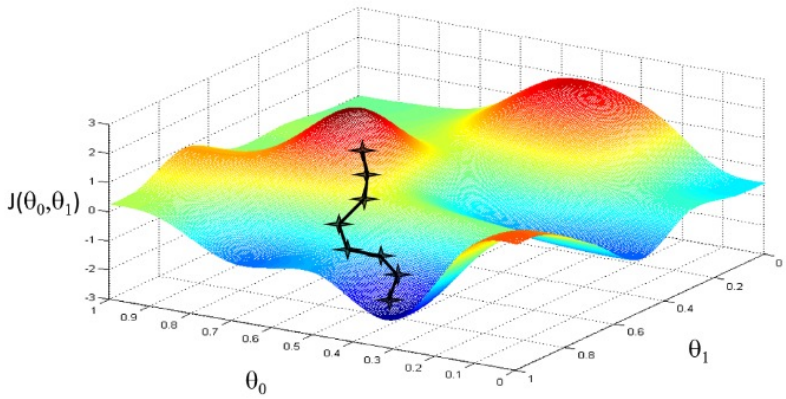
$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \xrightarrow{\mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{c}} \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{\mathrm{ReLU}} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{\mathbf{w}^{\mathrm{T}}\mathbf{h} + b} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$\mathbf{X}^{\mathrm{T}}$

$\mathbf{h}$

Gradient-Based Learning

- Nonlinearity of NN causes non-convex loss functions.
- NNs are usually trained by using iterative, gradient-based optimizers, such as stochastic gradient descent methods.
  - No convergence guarantee
  - Sensitive to initial values of parameters
    - Weights→small random values
    - Bias→zero or small positive values
  - Issues:
    - Cost (loss) functions
    - Computation of gradients
    - Gradient-based optimization



Cost Functions: Maximum Likelihood

- Maximum likelihood model for the distribution of output *y*:

$$p(\mathbf{y}|\mathbf{x};\theta)$$

Leopard cat or kitty?

- Cost function is negative log-likelihood, or cross-entropy between the training data and the model distribution:

$$J(\theta) = -E_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y}|\mathbf{x};\theta)$$

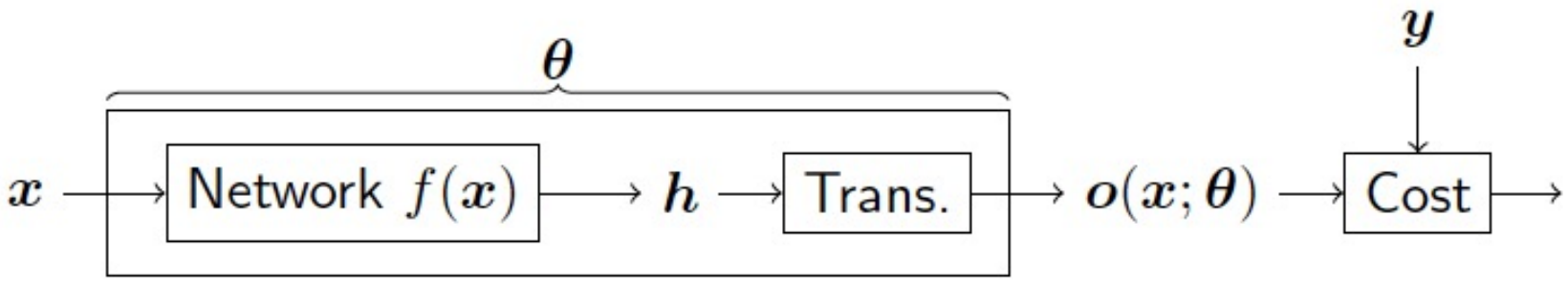
  - If $p_{\text{model}}(\mathbf{y}|\mathbf{x};\theta) = N(\mathbf{y}; f(\mathbf{x},\theta), \mathbf{I})$,

$$J(\theta) = \frac{1}{2} E_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2 + \text{const}$$

    - Mean squared error cost
  - Large gradient and predictable for learning are preferable.
    - Many output units involve an *exp* function that can saturate when its argument is very negative.
    - *log* in cost function can undoes the *exp* functions

Gradient-based Learning

- General setting



  - $\boldsymbol{x}$: Inputs
  - $f(\boldsymbol{x})$: Feedforward network
  - $\boldsymbol{h}$: Hidden units computed by $f(\boldsymbol{x})$
  - Trans.: Output layer transforming $\boldsymbol{h}$ to output $\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta})$
  - $\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta})$: Output units parameterized by model parameters $\boldsymbol{\theta}$
  - Cost: A function of ground-truth $\boldsymbol{y}$ and output $\boldsymbol{o}$ to be minimized w.r.t. model parameters $\boldsymbol{\theta}$

Cost Functions

- The maximum likelihood (ML) principle provides a guide for designing cost functions
- If we define a conditional distribution $p(y|\boldsymbol{x})$ as a distribution over y parameterized by the network outputs $\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta})$

$$p(y|\boldsymbol{x}) \triangleq p(y;\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta}))$$

- Then, the ML principle suggests we take the negative log-likelihood

$$-\log p(y;\boldsymbol{o}(\boldsymbol{x};\boldsymbol{\theta}))$$

as the cost function to be minimized w.r.t. model parameters $\boldsymbol{\theta}$

Learning Conditional Statistics

- If we define

$$p(y|\boldsymbol{x}) \triangleq \mathcal{N}(y; o(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{I}),$$

- Then, minimizing the negative log-likelihood yields

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} E_{\boldsymbol{x},y\sim p_{\text{data}}} \|y - o(\boldsymbol{x}; \boldsymbol{\theta})\|^2$$

- With sufficient capacity, the network will learn the conditional mean

$$o(\boldsymbol{x}; \boldsymbol{\theta}^*) \approx E[y|\boldsymbol{x}],$$

which predicts the mean value of y for each $\boldsymbol{x}$

- By the same token, if we define

$$p(y|\boldsymbol{x}) \triangleq \text{Laplace}(y; o(\boldsymbol{x};\boldsymbol{\theta}), \gamma) = \frac{1}{2\gamma} exp\left(-\frac{|y - o(\boldsymbol{x};\boldsymbol{\theta})|}{\gamma}\right),$$

- Then, minimizing the negative log-likelihood yields

$$\boldsymbol{\theta^*} = \arg\min_{\boldsymbol{\theta}} E_{\boldsymbol{x},y\sim p_{\text{data}}} |y - o(\boldsymbol{x};\boldsymbol{\theta})|$$

- With sufficient capacity, the network will learn the conditional median

$$o(\boldsymbol{x};\boldsymbol{\theta}^*) \approx \text{Median}[y|\boldsymbol{x}],$$

  which predicts the median value of y for each $\boldsymbol{x}$

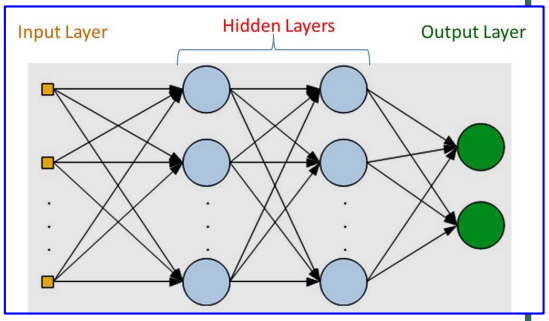
- In other words, $o(\boldsymbol{x};\boldsymbol{\theta}^*)$ satisfies

$$\int_{-\infty}^{o(\boldsymbol{x};\boldsymbol{\theta}^*)} p(y|\boldsymbol{x})dy = \int_{o(\boldsymbol{x};\boldsymbol{\theta}^*)}^{\infty} p(y|\boldsymbol{x})dy$$

Output Units

- The choice of cost function is tightly coupled with the choice of output unit.
- Three common output units: linear, sigmoid, and softmax.
- Linear units for Gaussian output distributions
  - Given features ***h***, output units produce a vector:

$$\hat{\mathbf{y}} = \mathbf{W}^T\mathbf{h} + \mathbf{b}$$



  - Linear units are often used to produce the mean of a conditional Gaussian distribution:

$$p(\mathbf{y}|\mathbf{x}) = N(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I})$$

  - Maximizing the log-likelihood is equivalent to minimizing the mean squared error.
  - Linear units do not saturate.

Learning Gaussian Output Distributions

- The n-dimensional Gaussian distribution $\mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is given by

$$\mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp(-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{y}-\boldsymbol{u}))$$

- A conditional Gaussian can be learned by treating $o(\boldsymbol{x}; \boldsymbol{\theta})$ as the means

$$p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; o(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{I})$$

- In this model, the outputs $o(\boldsymbol{x}; \boldsymbol{\theta})$ often take a linear form

$$o(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{W}^T \boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{b}$$

- As such, they are known as *linear output units*

Output Units

- Sigmoid units for Bernoulli output distributions
  - Used to predict the value of a binary variable *y*, in classification problems with two classes.
  - To predict $P(y = 1|\mathbf{x})$, which lies in the interval [0,1].
  - Simply use a linear unit followed by thresholding:

$$P(y = 1|\mathbf{x}) = \max\{0, \min\{1, \mathbf{w}^T\mathbf{h} + b\}\}$$

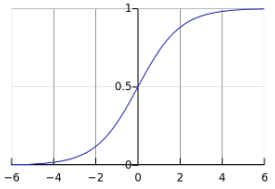
    - Not good, because the gradient would be 0 when the linear output stray outside the unit interval.

Output Units

- Sigmoid units for Bernoulli output distributions
  - Use a logistic sigmoid output combined with maximum likelihood:

$$\hat{y} = \sigma(z) = \sigma(\mathbf{w}^T\mathbf{h} + b), z = \mathbf{w}^T\mathbf{h} + b$$

y=0 ⟵ ⟶ y=1



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

  - To define a probability distribution over *y* using *z*:

$$\log \tilde{P}(y) = yz, \quad y \in \{0, 1\}$$

$$\tilde{P}(y) = e^{yz}$$

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

$$P(y) = \sigma((2y-1)z), \; z : \text{logit}$$

$$P(y=0) = \sigma(-z)$$
$$P(y=1) = \sigma(z)$$

Output Units

- Sigmoid units for Bernoulli output distributions
  - The loss function for maximum likelihood learning of a Bernoulli parameterized by a sigmoid is:

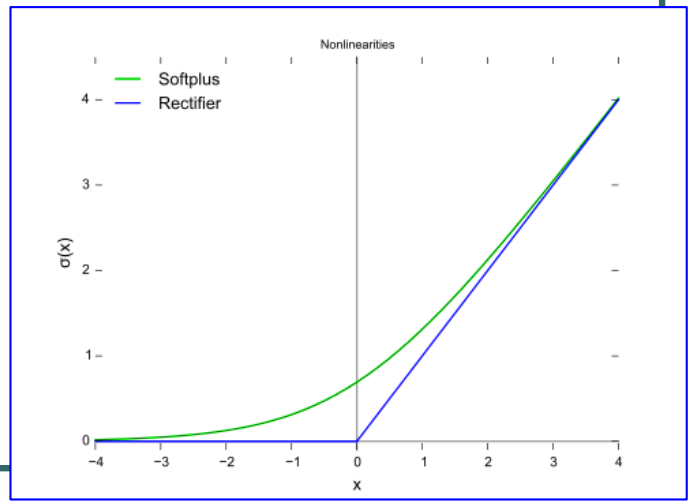
$$J(\theta) = -\log P(y|\mathbf{x}) = -\log \sigma((2y-1)z)) = \zeta((1-2y)z)$$

$$\zeta(x) = \log(1+e^x) : \text{ softplus function}$$

$$y \in \{0, 1\}$$

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

  - The softplus function does not shrink the gradient:
    - It saturates only when the answer is right (y=1, z is positive or y=0, z is negative), such that (1-2*y*)*z* is negative.
    - When the answer is wrong, the softplus function returns ~|*z*|



Output Units

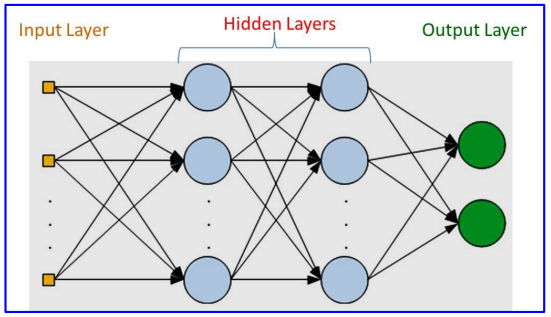
- Sigmoid units for Bernoulli output distributions
  - When sigmoid units are combined with other cost functions like mean squared error, the gradient vanishing problem may occur.
  - The choice of output units is tightly coupled with that of cost functions.

Output Units

- Softmax units for Multinoulli output distributions
  - A probability distribution over a discrete variable with *n* possible values
  - Used as the output of a classifier for *n* classes
  - Goal: a vector $\hat{\mathbf{y}}$, $\hat{y}_i = P(y = i|\mathbf{x})$
    - Each $\hat{y}_i$ lies in the interval [0,1].
    - Entire vector $\hat{\mathbf{y}}$ sums to 1.

$$\mathbf{z} = \mathbf{W}^T\mathbf{h} + \mathbf{b},\ z_i = \log \tilde{P}(y = i|\mathbf{x})$$

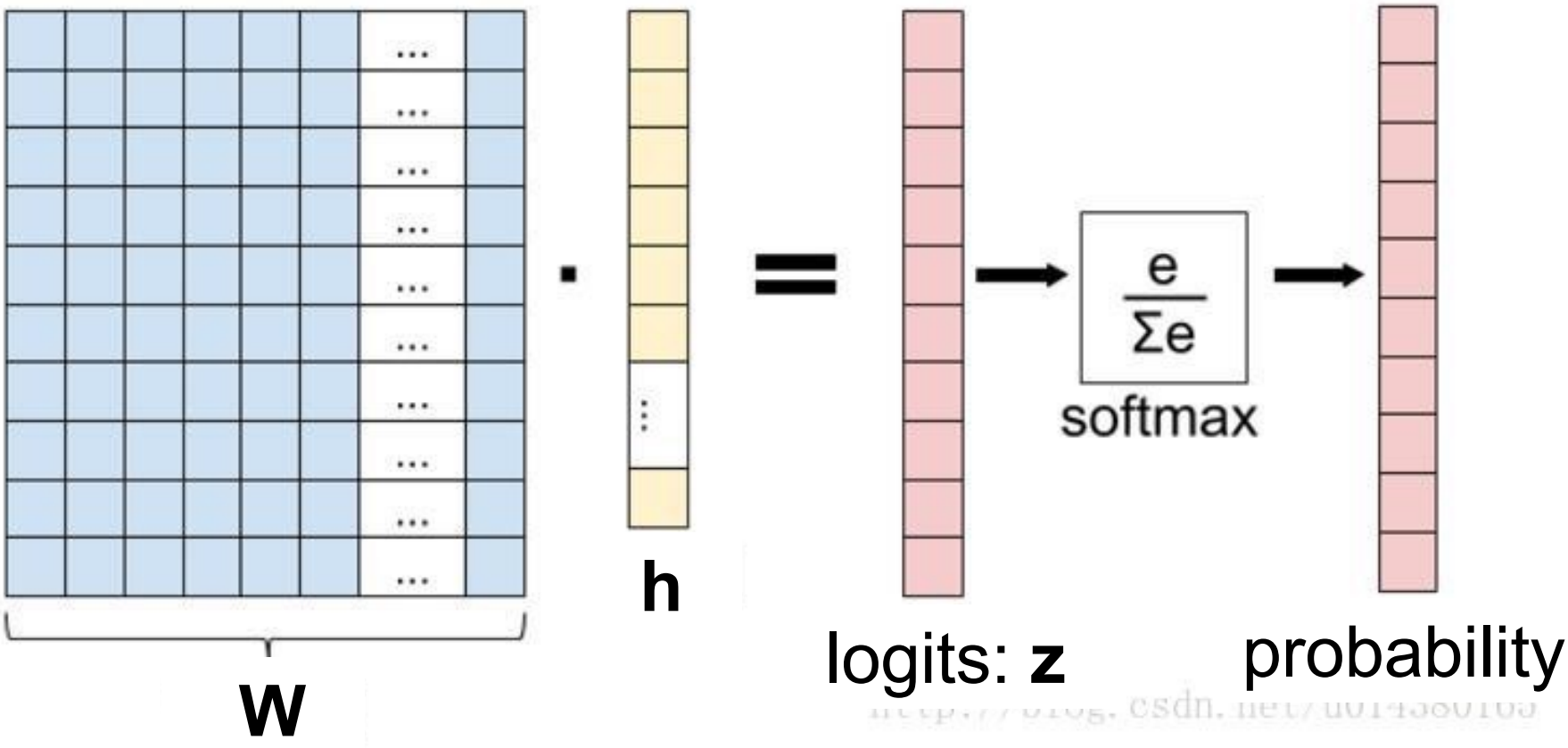
$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$



In Sigmoid unit: $P(y = 1) = \frac{e^z}{\sum_{y'=0}^{1} e^{y'z}}$

Output Units

- Softmax units for Multinoulli output distributions



Output Units

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

- Softmax units for Multinoulli output distributions
  - Minimize negative log-likelihood:

$$-\log P(y = i; \mathbf{z}) = -\log \text{softmax}(z)_i = -(z_i - \log \sum_{j=1}^{n} e^{z_j})$$

  - Input z_i always has a direct contribution to the cost function and cannot saturate.
  - While minimizing, the first term encourages z_i to be pushed up, while the second term encourages all of $\boldsymbol{z}$ to be pushed down.
  - For the correctly classified sample,

$$z_i \gg z_k, k \neq i \rightarrow z_i - \log \sum_{j=1}^{n} e^{z_j} \approx z_i - z_i = 0$$

  - Negative log-likelihood cost function always strongly penalizes the most active incorrect prediction.
  - Most objective function other than the negative log-likelihood do not work well with the softmax function.

Output Units

- Softmax units for Multinoulli output distributions
  - Softmax function can be easily out of range for extreme input values.

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

  - Note that $\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} + c)$
  - We can derive a numerically stable variant of softmax:

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} - \max_j z_j)$$

    - An output $\text{softmax}(\mathbf{z})_i$ saturates to 1 when the corresponding input z_i is maximal and much greater than all other inputs.
    - An output $\text{softmax}(\mathbf{z})_i$ can also saturate to 0 when z_i is not maximal and the maximum is much greater.

Output Units

- Softmax units for Multinoulli output distributions
  - Softmax is a way to create a form of competition between the units that participate in it.
  - From a neuroscience point of view, lateral inhibition is believed to exist between nearby neurons, that is, winner-take-all.
  - Softmax
    - softened version of arg max
    - Continuous and differentiable

Nature Neuroscience **2**, 375 - 381 (1999)
doi:10.1038/7286

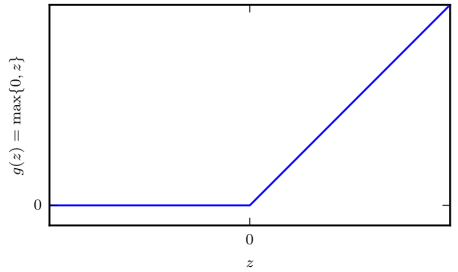
Attention activates winner-take-all competition among visual filters

D.K. Lee[1,2], L. Itti[1,2], C. Koch[1] & J. Braun[1]

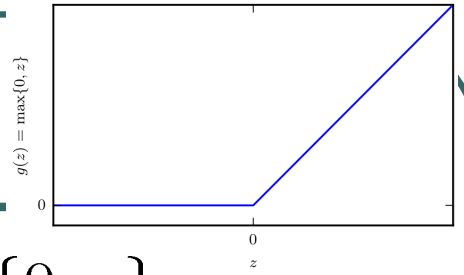
Shifting attention away from a visual stimulus reduces, but does not abolish, visual discrimination performance. This residual vision with 'poor' attention can be compared to normal vision with 'full' attention to reveal how attention alters visual perception. We report large differences between residual and normal visual thresholds for discriminating the orientation or spatial frequency of simple patterns, and smaller differences for discriminating contrast. A computational model, in which attention activates a winner-take-all competition among overlapping visual filters, quantitatively accounts for all observations. Our model predicts that the effects of attention on visual cortical neurons include increased contrast gain as well as sharper tuning to orientation and spatial frequency.

Hidden Units

- Design of hidden units in an extremely active area of research.
- ReLUs are an excellent default choice.
- Although not differentiable at all point, it is still okay to use for gradient-based learning algorithm.
  - Use left or right derivative, instead.
- Hidden units compute:
  - An affine transformation $\mathbf{z} = \mathbf{W}^T\mathbf{x} + \mathbf{b}$
  - An element-wise nonlinear function $g(\boldsymbol{z})$



ReLUs



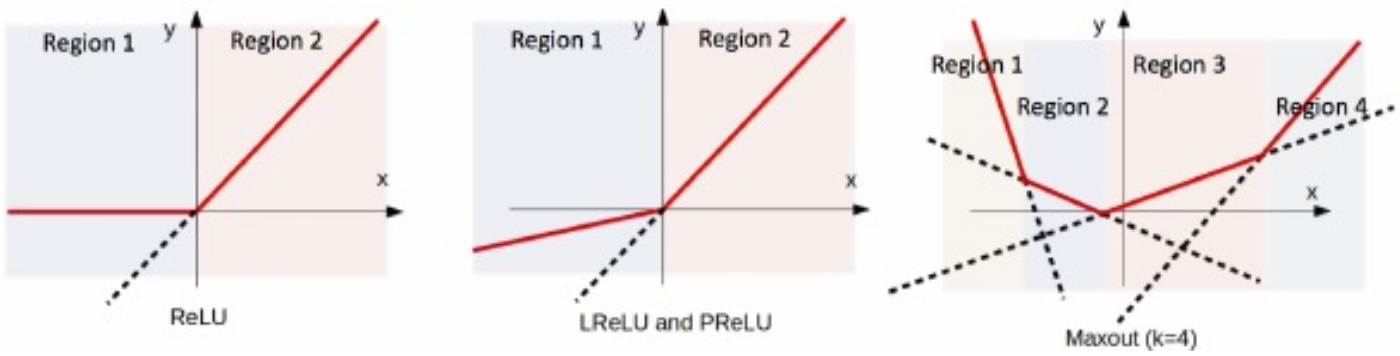
- ReLU: rectified linear unit: $g(z) = \max\{0, z\}$
- Easy to optimize because the derivatives through ReLU remain large whenever the unit is active.
- When initializing the parameters of the affine transformation, it can be a good practice to set all elements of ***b*** to a small, positive value, such as 0.1. (why?) $\mathbf{h} = g(\mathbf{W}^T\mathbf{x} + \mathbf{b})$
- Drawback: ReLUs cannot learn via gradient-based methods on examples with zero activation.

Generalizations of ReLUs

- Use a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\mathbf{z}, \alpha)_i = \max\{0, z_i\} + \alpha_i \min\{0, z_i\}$$

- Absolute value rectification fixes $\alpha_i = -1$: $g(z) = |z|$
- Leaky ReLU fixes α_i to a small value like 0.01.
- Parametric ReLU or PReLU treats α_i as a learnable parameter.

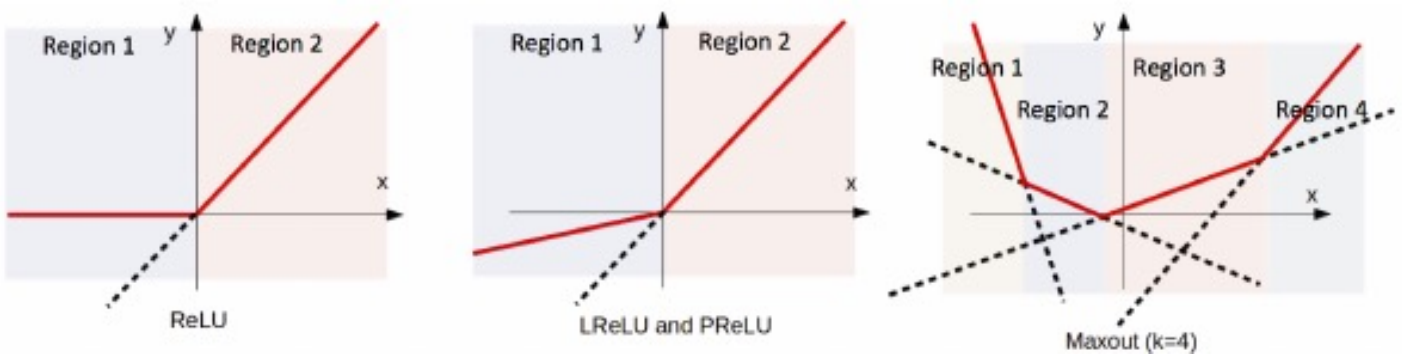


Generalizations of ReLUs

- Maxout units:

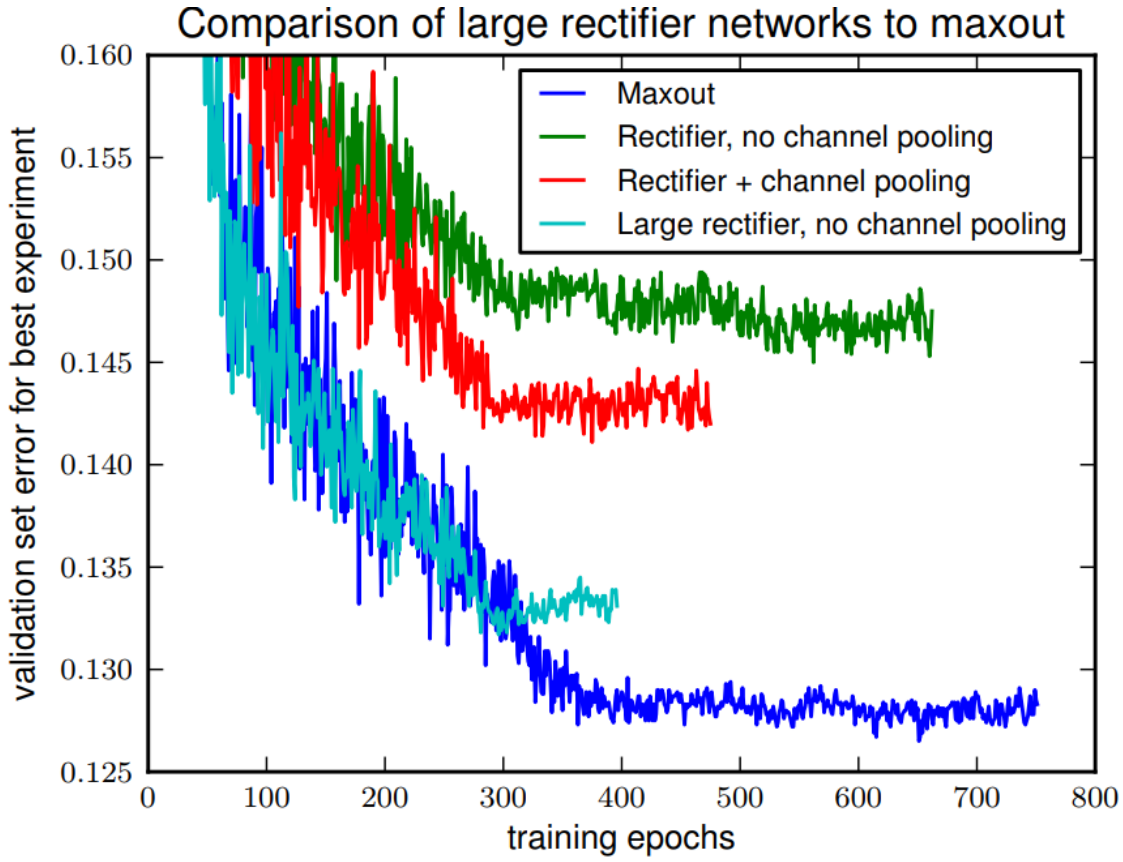
$$g(\mathbf{z})_i = \max_{j \in [1,k]} z_{ij} = \max\{\mathbf{w}_1^{\mathrm{T}}\mathbf{x} + \mathbf{b}_1, \mathbf{w}_2^{\mathrm{T}}\mathbf{x} + \mathbf{b}_2, \ldots, \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + \mathbf{b}_k\}$$

  - It becomes an ReLU when k=2, $\boldsymbol{w}_1$=$\boldsymbol{b}_1$=0
  - It can learn a piecewise linear, convex activation function with up to *k* pieces.



Maxout Units

- Maxout unit learns activation function
- Maxout unit improves abstraction ability



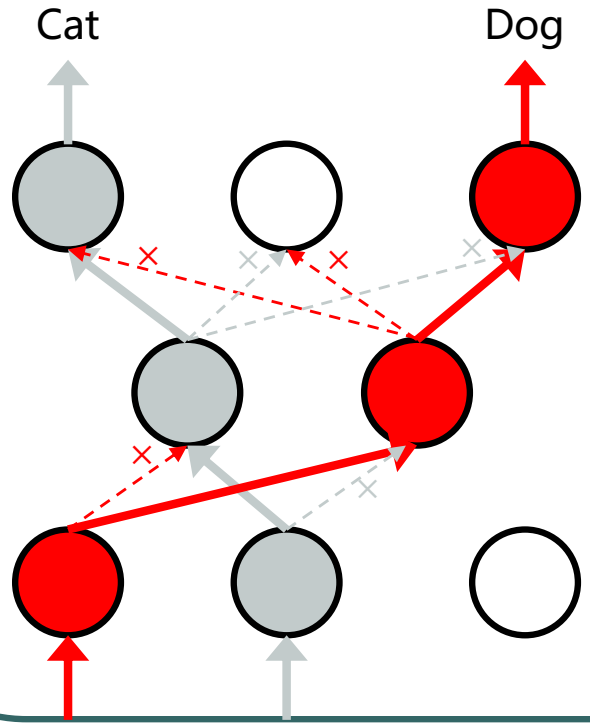
Goodfellow et al. *ICML* 2013

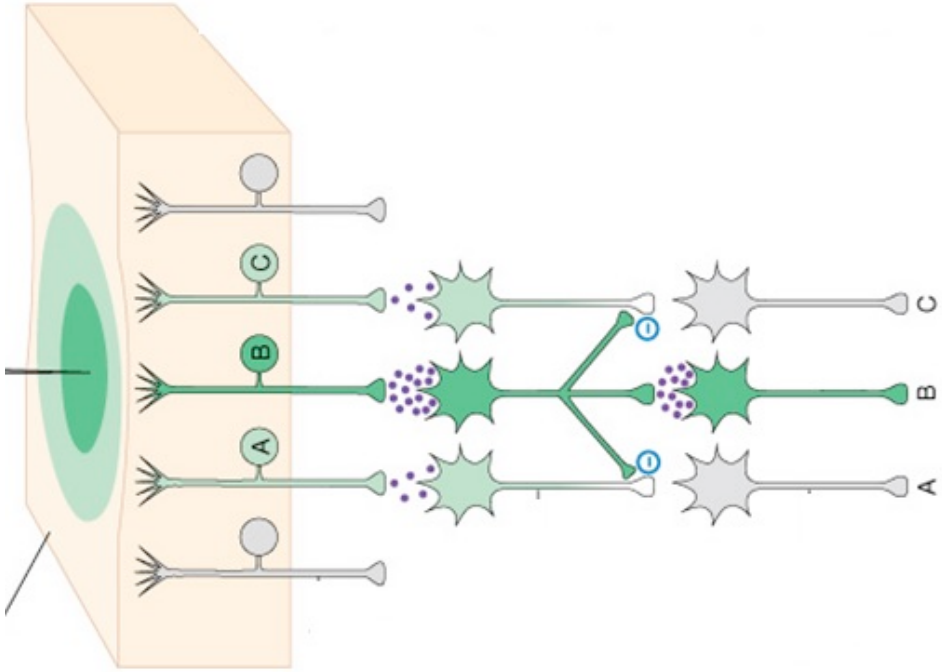
Maxout Inspired by Lateral Inhibition

- The **pathway** along which the signal flows that determines the *functionality* of information processing

  (Kandel et al. Principles of neural science)

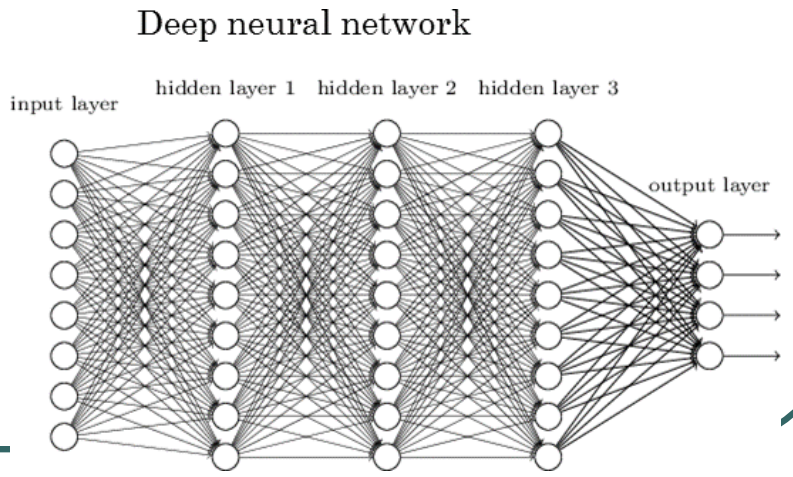
- Lateral inhibition





Architecture Design

- Architecture: overall structure of the network
  - How many units it should have
  - How these units should be connected to each other
  - How to choose the depth and width of each layer
- Deeper networks often:
  - Use far fewer units per layer and far fewer parameters
  - Generalize to the test set
  - Are harder to optimize

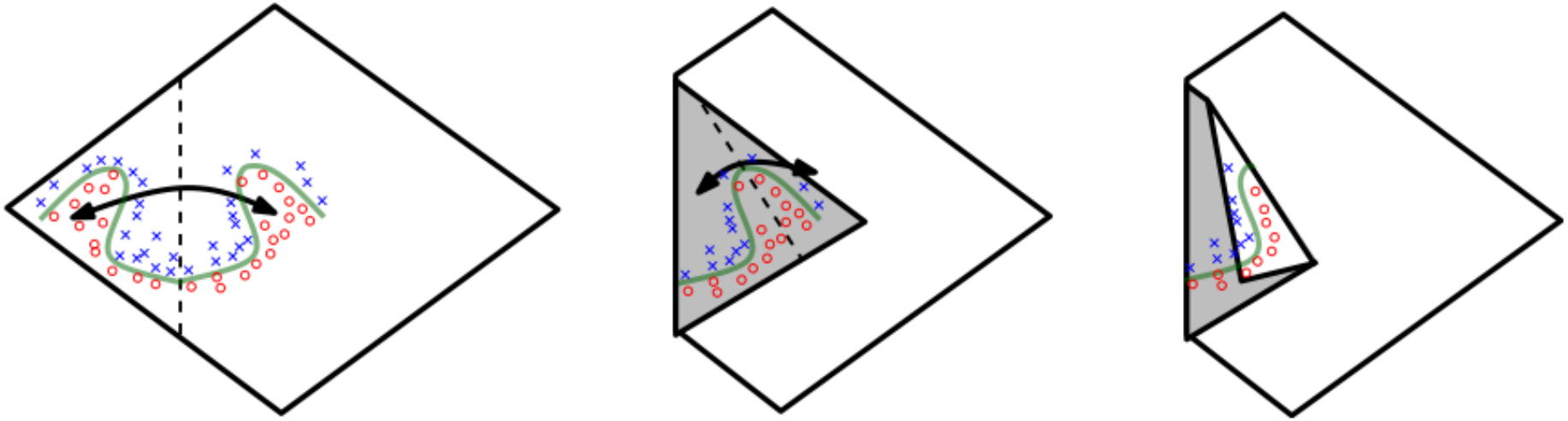


Universal Approximation Properties and Depth

- The universal approximation theorem states that a feedforward network with one linear output layer and at least one hidden layer with any "squashing" activation function (e.g., sigmoid) can approximate any Borel measurable function, provided that the network is given enough hidden units.
- Any continuous function $f : \Re^n \rightarrow \Re^m$ on a closed and bounded subset of $\Re^n$ is Borel measurable.
- A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.
- Using deeper models can reduce the number of units required to represent the desired function and can reduce the generalization error.

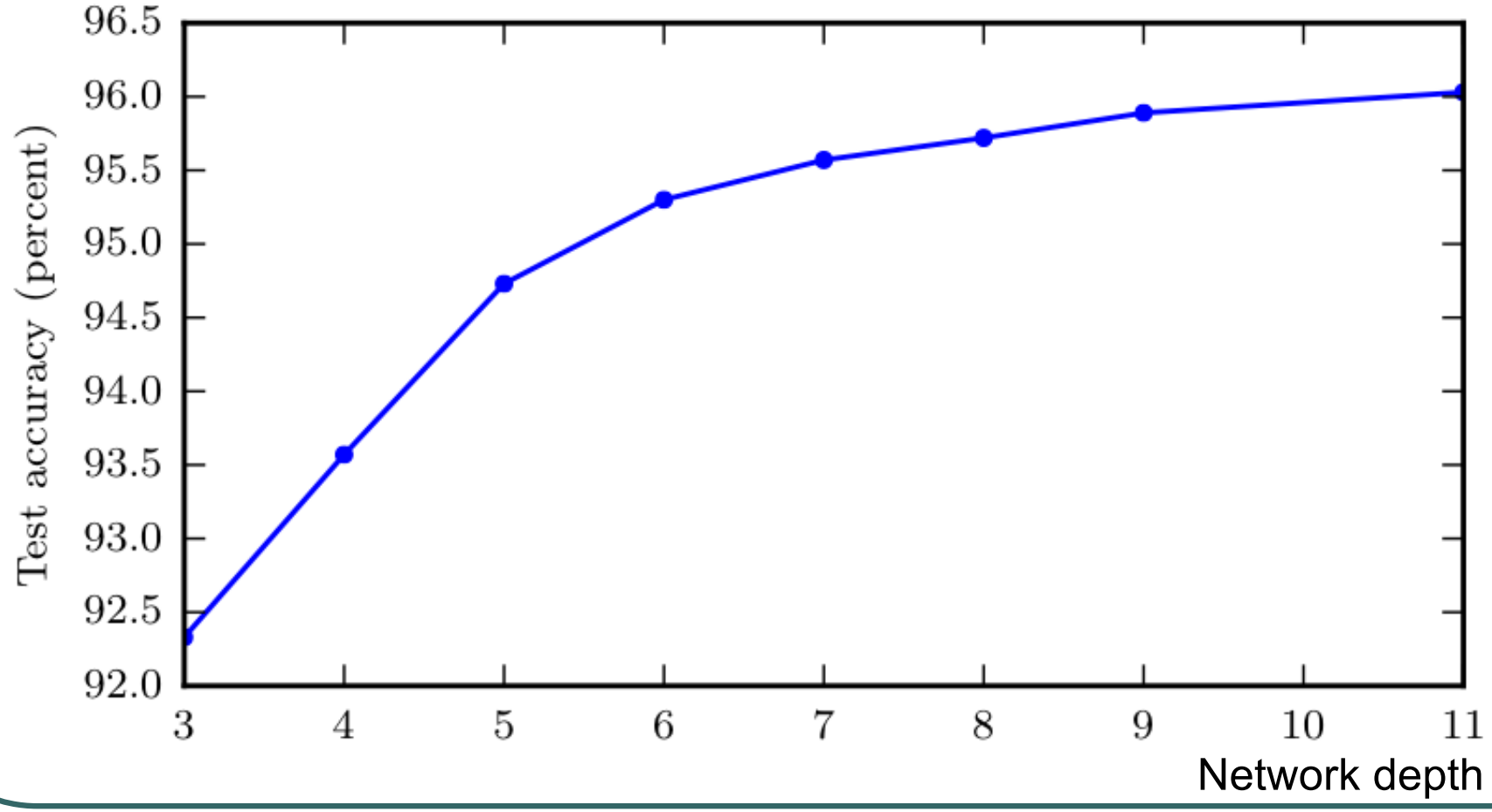
Universal Approximation Properties and Depth

- A network with absolute value rectification creates mirror images of the function computed on top of hidden units.

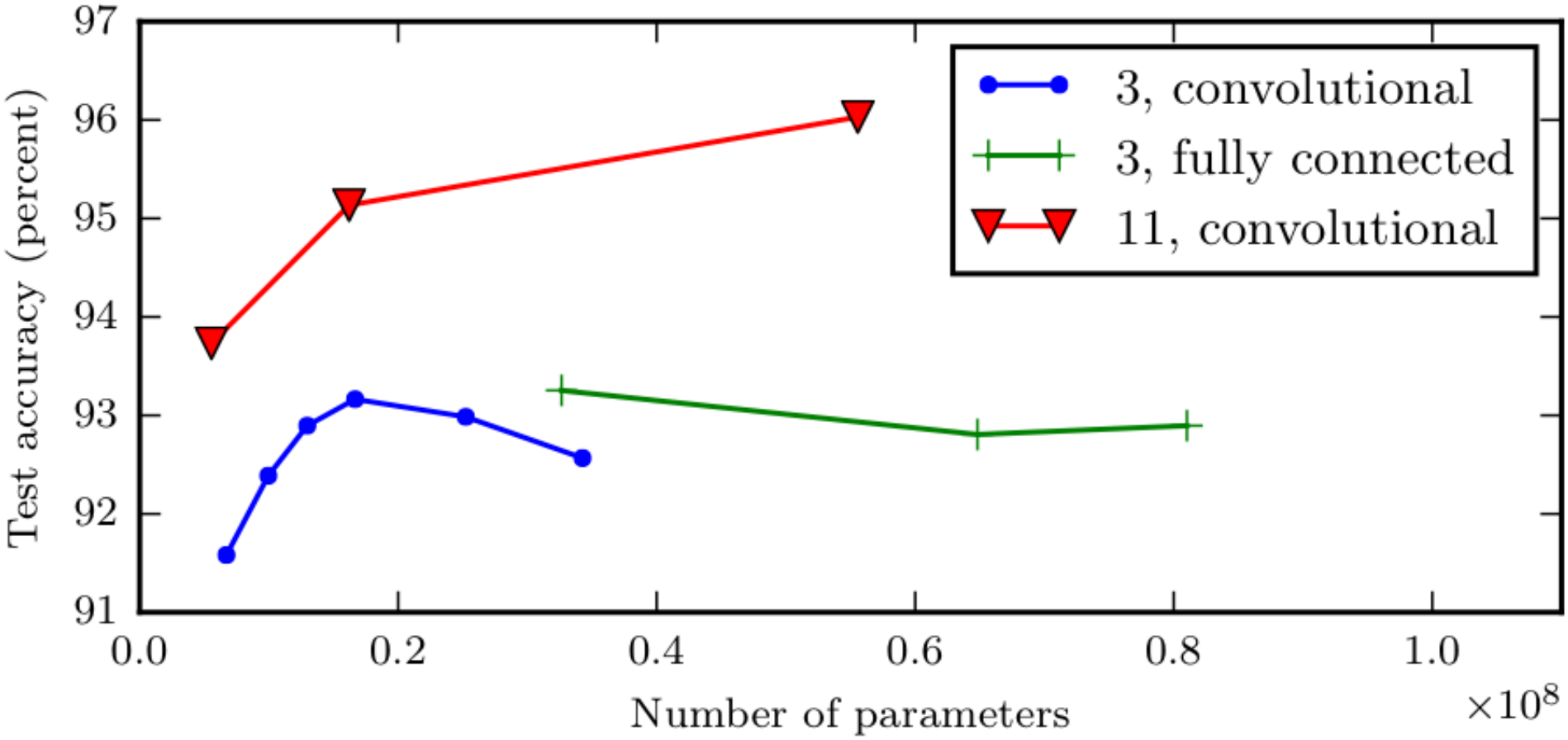


Better Generalization with Greater Depth

- Empirically, deeper networks generalize better.
- SVHN



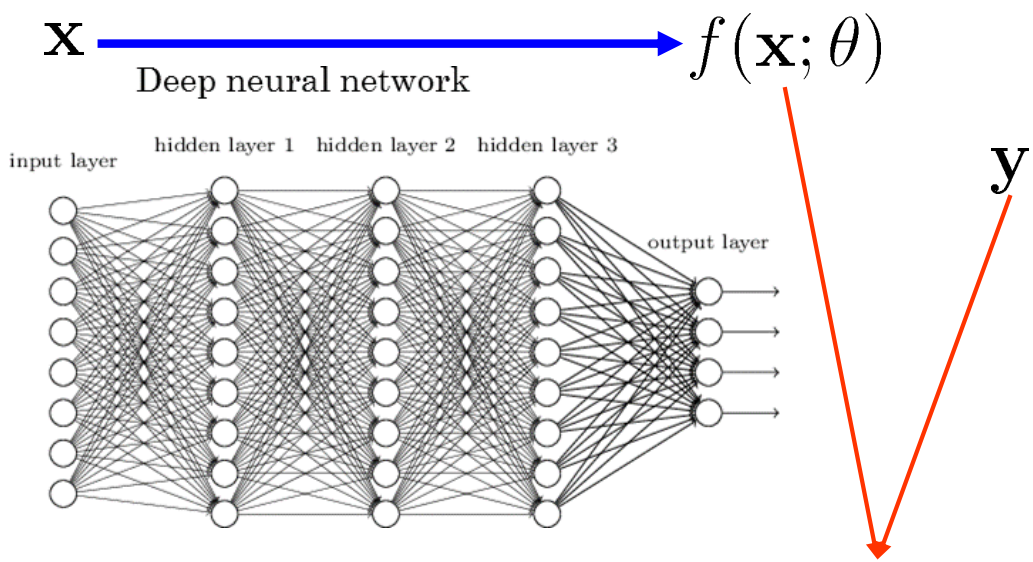
Large, Shallow Models Overfit More



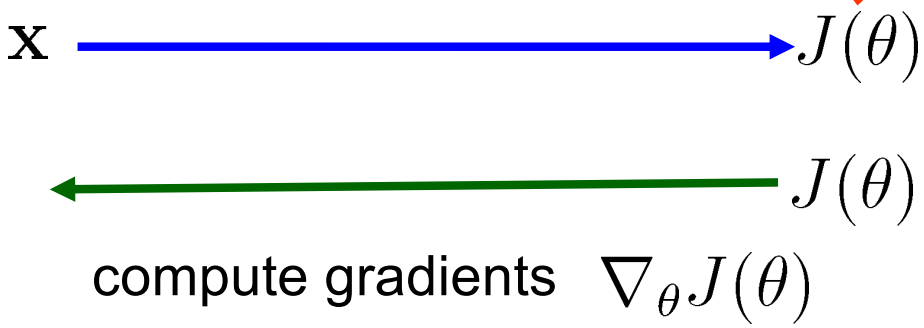
Back-Propagation

- During inference:

  (forward propagation)



- During training:
- Backpropagation:

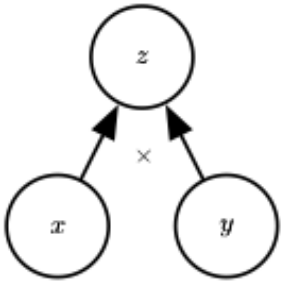


- Stochastic gradient descent is used to perform the learning using these gradients.

Computational Graphs

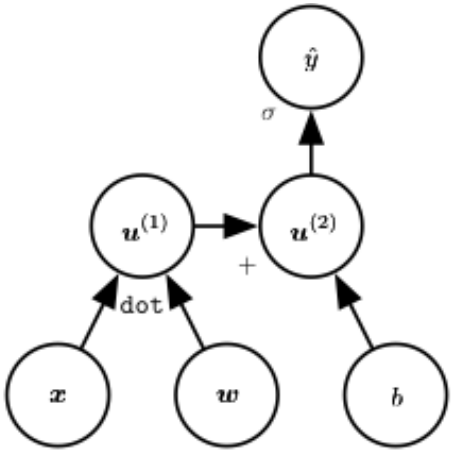
- Each node indicate a variable
  - Scalar
  - Vector
  - Matrix
  - Tensor

    (multi-dimensional array)

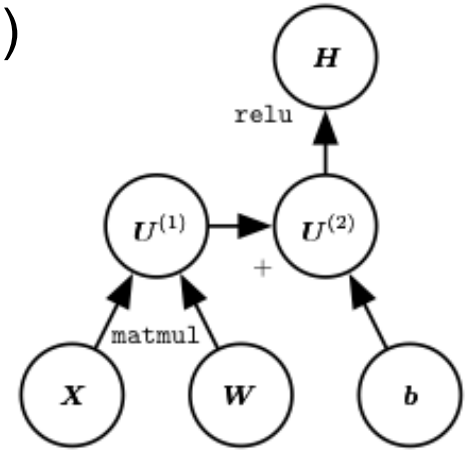


$$z = xy$$

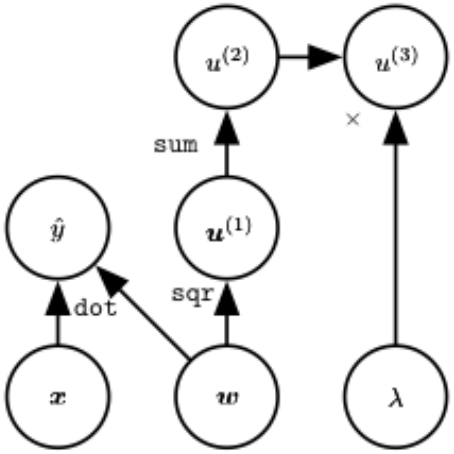
(a)



$$\hat{y} = \mathbf{w}^T\mathbf{x} + b$$



$$\mathbf{H} = \max\{0, \mathbf{W}^T\mathbf{X} + \mathbf{b}\}$$



(d)

Chain Rule of Calculus

- To compute the derivative dz/dx of a function $z(x)$ formed by the composition of functions $z(x) = f(g(x))$

$$x \xrightarrow{g(x)} y \xrightarrow{f(y)} z$$

  - $x \in \mathbb{R}$ is a real number
  - $f, g : \mathbb{R} \to \mathbb{R}$ are real-valued functions of single variable

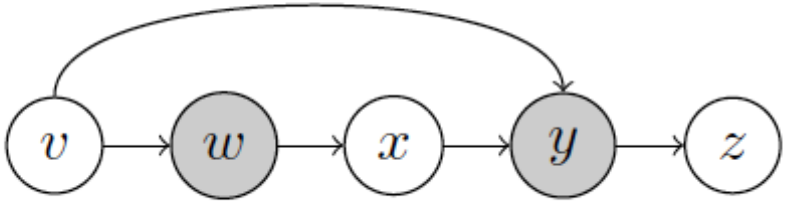
- The chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- As an extension, we have for the following graph

$$\frac{dz}{dv} = \frac{dz}{dy}\frac{dy}{dv} + \frac{dz}{dw}\frac{dw}{dv} = \sum_{n:v\in Pa(n)} \frac{dz}{dn}\frac{dn}{dv}$$

where $Pa(n)$ is the set of nodes that are parents of n



- To verify the result requires another chain rule from calculus

$$z = f(y_1, y_2),\ y_1 = g_1(x),\ y_2 = g_2(x)$$

$$\frac{dz}{dx} = \frac{dz}{dy_1}\frac{dy_1}{dx} + \frac{dz}{dy_2}\frac{dy_2}{dx}$$

Vector Case

- z is a scalar, and $\boldsymbol{x}, \boldsymbol{y}$ are vectors

$$\boldsymbol{x}_{m\times 1} \xrightarrow{g(\boldsymbol{x})} \boldsymbol{y}_{n\times 1} \xrightarrow{f(\boldsymbol{y})} z_{1\times 1}$$

- Applying the chain rule leads to

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i},\ i = 1, 2, ..., m$$

- In matrix notation, $\nabla_{\mathbf{x}} z = (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T \nabla_{\mathbf{y}} z$ $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$: Jacobian matrix

$$\begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \\ \vdots \\ \frac{\partial z}{\partial y_n} \end{bmatrix}$$

- This is recognized as a **Jacobian-gradient product**

$$\nabla_{\boldsymbol{x}} z = (\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}})^T \nabla_{\boldsymbol{y}} z,$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ (abbrev. $J_{y,x}$) is known as **Jacobian matrix** with

$$(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}})_{i,j} = \frac{\partial y_i}{\partial x_j}$$

As an example, when $\boldsymbol{y} = g(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}$ (i.e. $y_i = \sum_j w_{i,j} x_j$),

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \boldsymbol{W}$$

Matrix Case

- z is a scalar, and $\boldsymbol{X}, \boldsymbol{Y}$ are matrices

$$\boldsymbol{X}_{m\times n} \xrightarrow{g(\boldsymbol{X})} \boldsymbol{Y}_{s\times k} \xrightarrow{f(\boldsymbol{Y})} z_{1\times 1}$$

- The chain rule suggests that

$$\frac{\partial z}{\partial x_{i,j}} = \sum_{s,k} \frac{\partial z}{\partial y_{s,k}} \frac{\partial y_{s,k}}{\partial x_{i,j}}, \ \forall i, j$$

- More generally, when $\boldsymbol{X}, \boldsymbol{Y}$ are tensors (high-dimensional arrays),

$$\nabla_{\boldsymbol{X}} z = \sum_j (\frac{\partial z}{\partial Y_j}) \nabla_{\boldsymbol{X}} Y_j,$$

where $\boldsymbol{Y} = g(\boldsymbol{X}), z = f(\boldsymbol{Y})$ and $\boldsymbol{X}$ is treated as if it were a vector

- **Example 1:** Assume $\boldsymbol{Y} = g(\boldsymbol{X}) = \boldsymbol{W}\boldsymbol{X}$
  - Let $\nabla_{\boldsymbol{Y}} z$ denote a matrix with its element (i, j) given by $\partial z / \partial Y_{i,j}$
  - And $\nabla_{\boldsymbol{X}} z$ be a matrix with its element (i, j) given by $\partial z / \partial X_{i,j}$
  - Observe that each column $\boldsymbol{Y}_{:,j}$ of $\boldsymbol{Y}$ is a function of the corresponding column $\boldsymbol{X}_{:,j}$ in $\boldsymbol{X}$, i.e. $\boldsymbol{Y}_{:,j} = \boldsymbol{W}\boldsymbol{X}_{:,j}$
  - We apply the Jacobian-gradient product of vector form to obtain

$$\nabla_{\boldsymbol{X}_{:,j}} z = \boldsymbol{W}^T \nabla_{\boldsymbol{Y}_{:,j}} z \Rightarrow \underbrace{\nabla_{\boldsymbol{X}} z = \boldsymbol{W}^T \nabla_{\boldsymbol{Y}} z}$$

- **Example 2:** Assume $\boldsymbol{Y} = g(\boldsymbol{X}) = \boldsymbol{X}\boldsymbol{W}$
  - Observe that each row $\boldsymbol{Y}_{i,:}$ of $\boldsymbol{Y}$ is a function of the corresponding row $\boldsymbol{X}_{i,:}$ in $\boldsymbol{X}$, i.e., $\boldsymbol{Y}_{i,:} = \boldsymbol{X}_{i,:}\boldsymbol{W}$, or equivalently, $\boldsymbol{Y}_{i,:}^T = \boldsymbol{W}^T \boldsymbol{X}_{i,:}^T$
  - Applying the Jacobian-gradient product of vector form yields

$$\nabla_{\boldsymbol{X}_{i,:}^T} z = \boldsymbol{W} \nabla_{\boldsymbol{Y}_{i,:}^T} z \Rightarrow \underbrace{\nabla_{\boldsymbol{X}} z = (\nabla_{\boldsymbol{Y}} z)\boldsymbol{W}^T}$$

Back-Propagation

- Toy problem: To compute the derivative of J w.r.t. x

$$x \xrightarrow{f(x)} z \xrightarrow{g(z)} h \xrightarrow{c(h)} J$$

- From the chain rule, we have

$$\begin{aligned}\frac{\partial J}{\partial x} &= \frac{\partial J}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial x} \\ &= c'(h)g'(z)f'(x) \\ &= c'(g(f(x)))g'(f(x))f'(x)\end{aligned}$$

- There are two possible implementations
  - The one based on the last equality incurs redundant subcomputation (e.g. $f(x)$)
  - The other following the penultimate equality requires z, h be pre-computed through forward propagation

Back-Propagation

- Toy problem: To compute the derivative of J w.r.t. x

$$x \xrightarrow{f(x)} z \xrightarrow{g(z)} h \xrightarrow{c(h)} J$$

- From the chain rule, we have

$$\begin{aligned}\frac{\partial J}{\partial x} &= \frac{\partial J}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial x} \\ &= c'(h)g'(z)f'(x)\end{aligned}$$

- Now, assuming z, h have been pre-computed, one way to compute the derivatives of J w.r.t. all variables x, z, h is to proceed in the order of

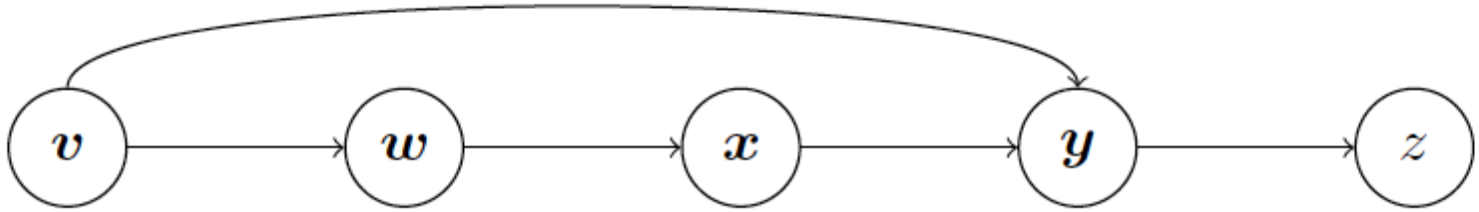
  1. $\partial J/\partial h = c'(h)$
  2. $\partial J/\partial z = (\partial J/\partial h)g'(z)$
  3. $\partial J/\partial x = (\partial J/\partial z)f'(x)$

  and, at each step, keep the result for subsequent use

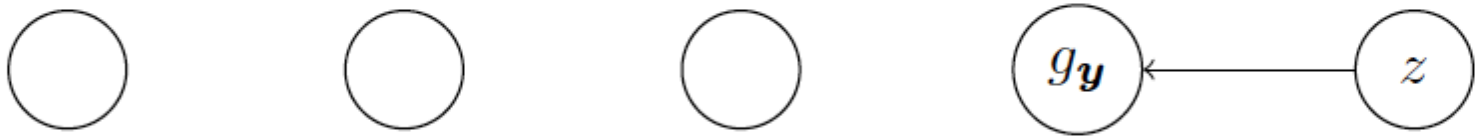
- This technique is known as the **back-propagation (backprop)** method

General Back-Propagation

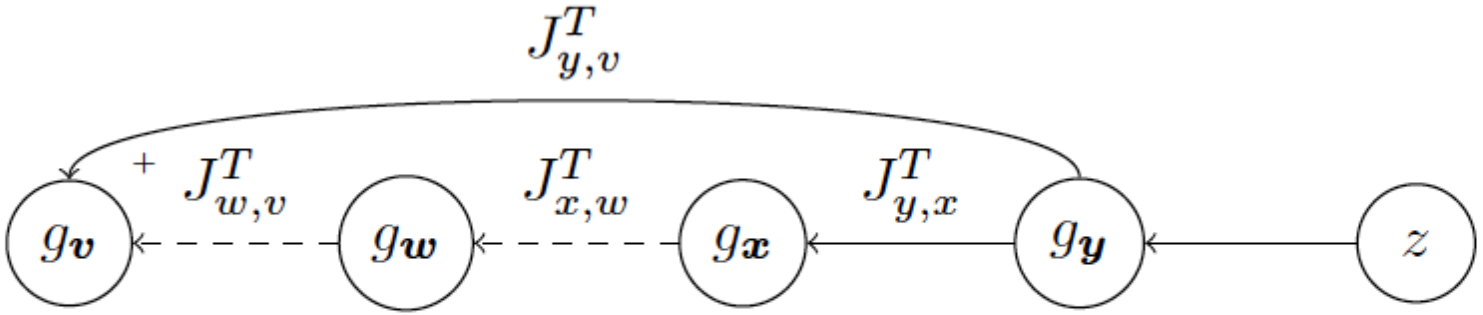
- To compute the gradient of z w.r.t. all its ancestors $\boldsymbol{y}, \boldsymbol{x}, \boldsymbol{w}, \boldsymbol{v}$



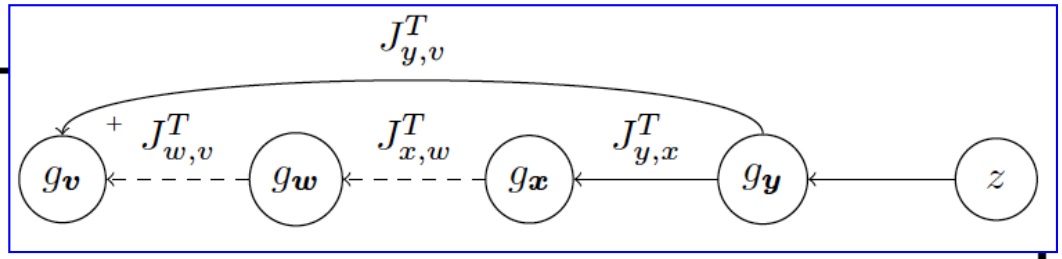
- Compute gradient w.r.t. every parent of z



- Travel backward, multiply current gradient by Jacobian recursively



- Sum gradients from different paths



- In symbols, we have

$$g_{\boldsymbol{y}} = \nabla_{\boldsymbol{y}} z$$

$$g_{\boldsymbol{x}} = \nabla_{\boldsymbol{x}} z = J^T_{y,x} g_{\boldsymbol{y}}$$

$$g_{\boldsymbol{w}} = \nabla_{\boldsymbol{w}} z = J^T_{x,w} g_{\boldsymbol{x}}$$

$$g_{\boldsymbol{v}} = \nabla_{\boldsymbol{v}} z = J^T_{w,v} g_{\boldsymbol{w}} + J^T_{y,v} g_{\boldsymbol{y}}$$

where

$$J_{y,x} = \partial \boldsymbol{y} / \partial \boldsymbol{x}$$

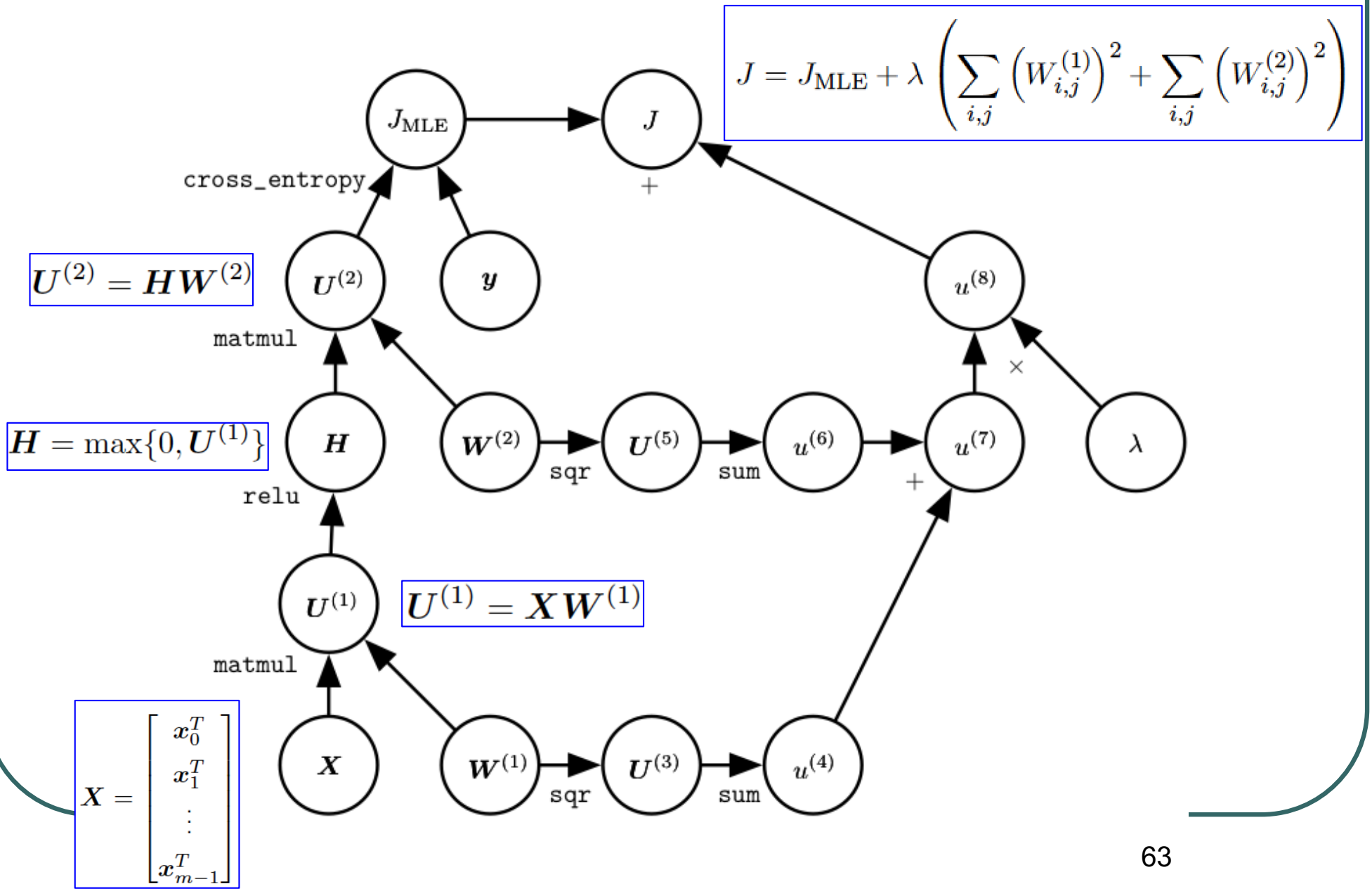
$$J_{x,w} = \partial \boldsymbol{x} / \partial \boldsymbol{w}$$

$$J_{w,v} = \partial \boldsymbol{w} / \partial \boldsymbol{v}$$

$$J_{y,v} = \partial \boldsymbol{y} / \partial \boldsymbol{v}$$

- The spirit of this procedure can extend to cases where $\boldsymbol{y}, \boldsymbol{x}, \boldsymbol{w}, \boldsymbol{v}$ are matrices, tensors, vectors, scalars, or their mixing combinations

Backprop for MLP Training



- Backprop: To compute $\nabla_{\boldsymbol{W}^{(1)}} J$ and $\nabla_{\boldsymbol{W}^{(2)}} J$
  - Two paths from J to the weights (only one path illustrated)
  - Assume $\nabla_{\boldsymbol{U}^{(2)}} J = \boldsymbol{G}$
  - Then $\nabla_{\boldsymbol{W}^{(2)}} J = \boldsymbol{H}^T \boldsymbol{G}$ (cf. **Example 1** in Matrix Case)
  - Similarly, $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)T}$ (cf. **Example 2** in Matrix Case)
  - Tracing back further, we have $\nabla_{\boldsymbol{U}^{(1)}} J = \boldsymbol{G}'$ by zeroing out elements in $\nabla_{\boldsymbol{H}} J$ corresponding to entries of $\boldsymbol{U}^{(1)}$ less than zero
  - Again, $\nabla_{\boldsymbol{W}^{(1)}} J = \boldsymbol{X}^T \boldsymbol{G}'$ (cf. **Example 1** in Matrix Case)

$\boldsymbol{U}^{(2)} = \boldsymbol{H}\boldsymbol{W}^{(2)}$

$\boldsymbol{U}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)}$