Deep Feedforward Networks

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

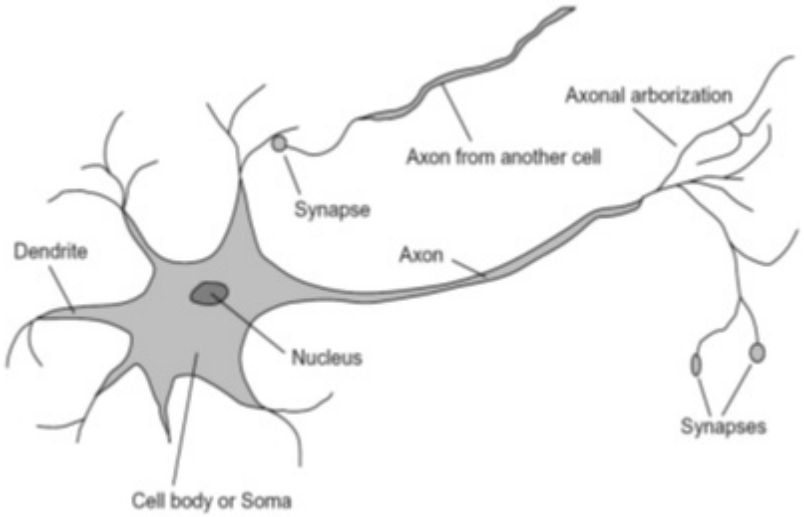
Sources:

"Neuroscience---exploring the brain," 2nd ed.
by Bear, Connors, and Paradiso

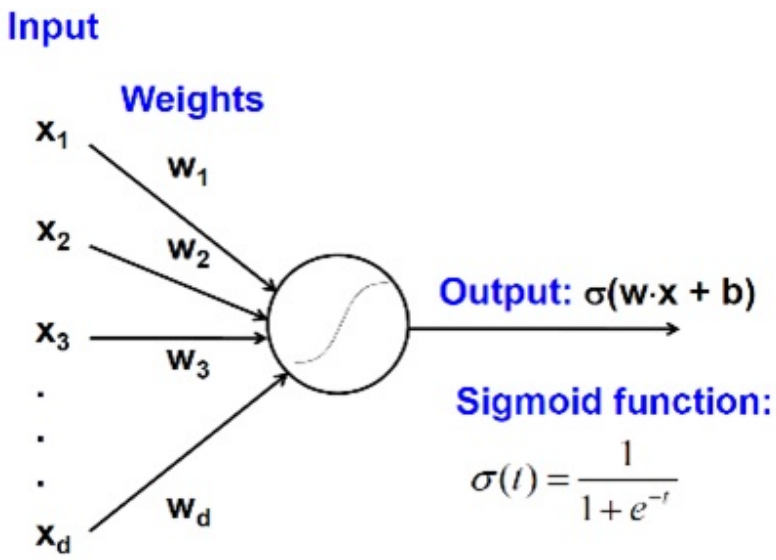
"CS 790: Introduction to Machine Learning" at U. Wisconsin
by Jia-Bin Huang, Virginia Tech.

Ch. 6, "Deep Learning" textbook
by Goodfellow et al.

Biological Neuron and Perceptrons



A biological neuron

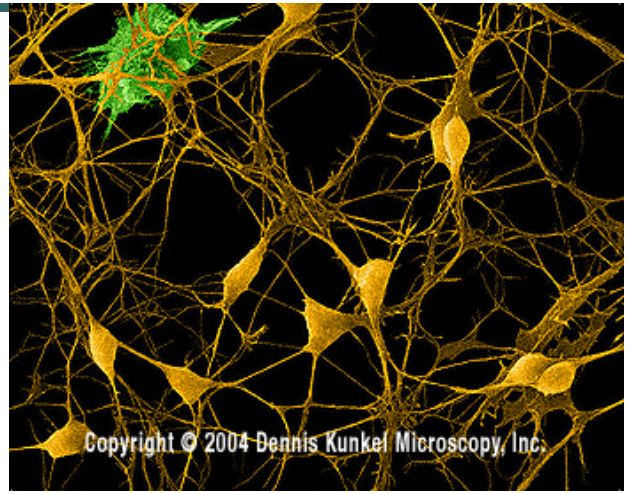


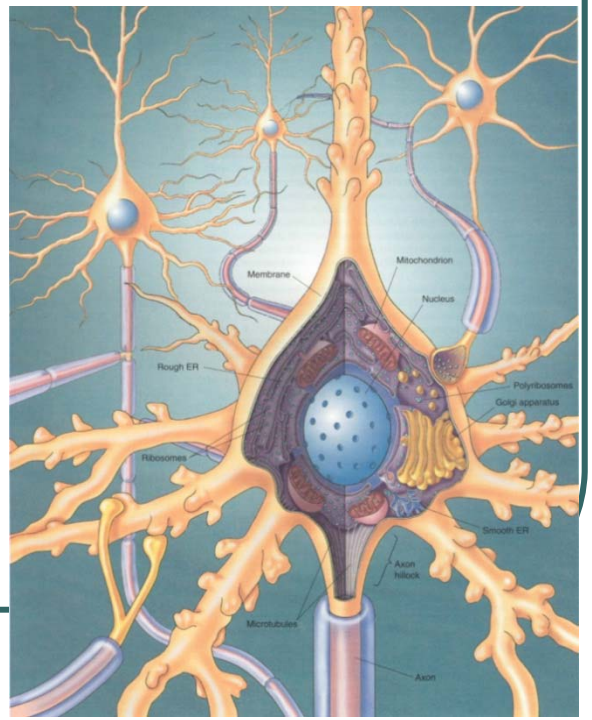
Frank Rosenblatt, 1957

An artificial neuron (Perceptron)
- a linear classifier

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

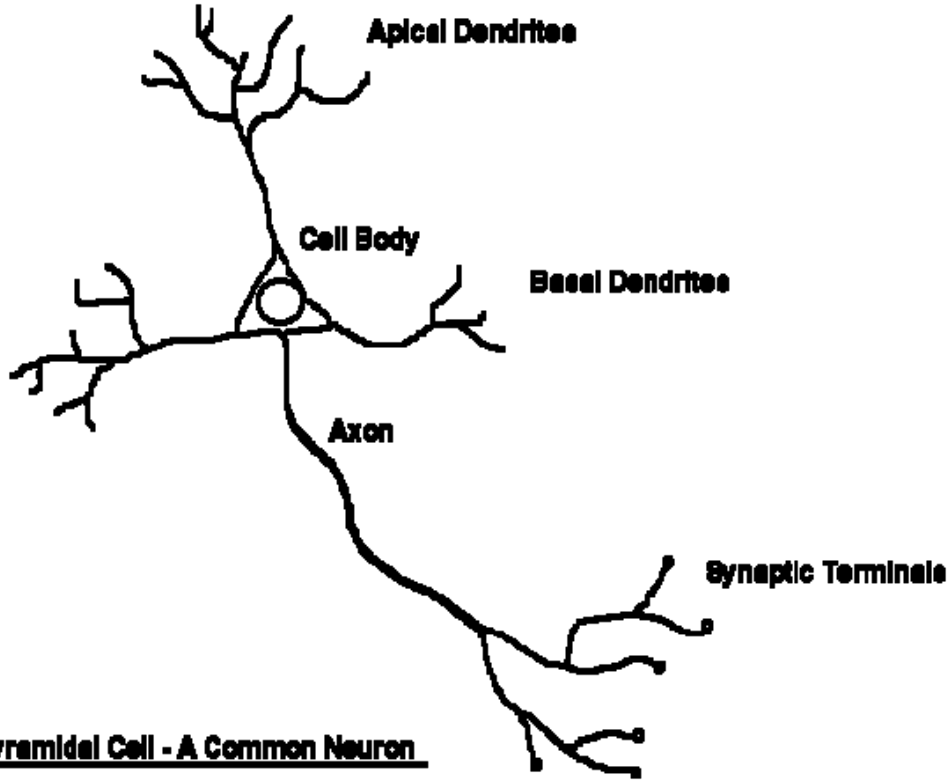




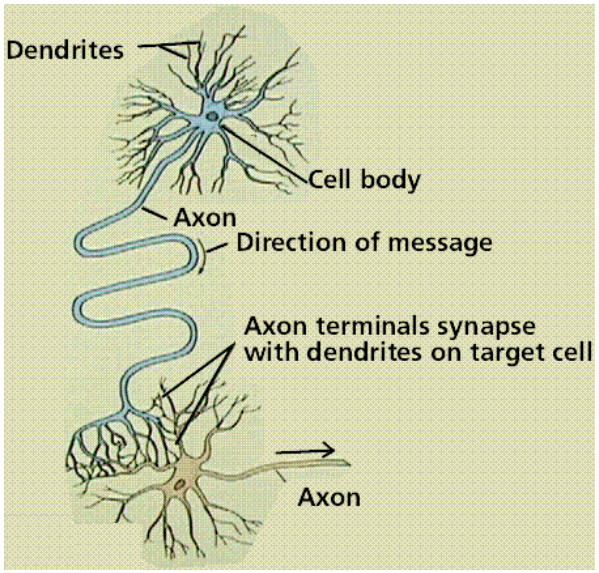
Connors et al. 2002, *Neuroscience: exploring the brain*

Structure of the Cerebral Cortex

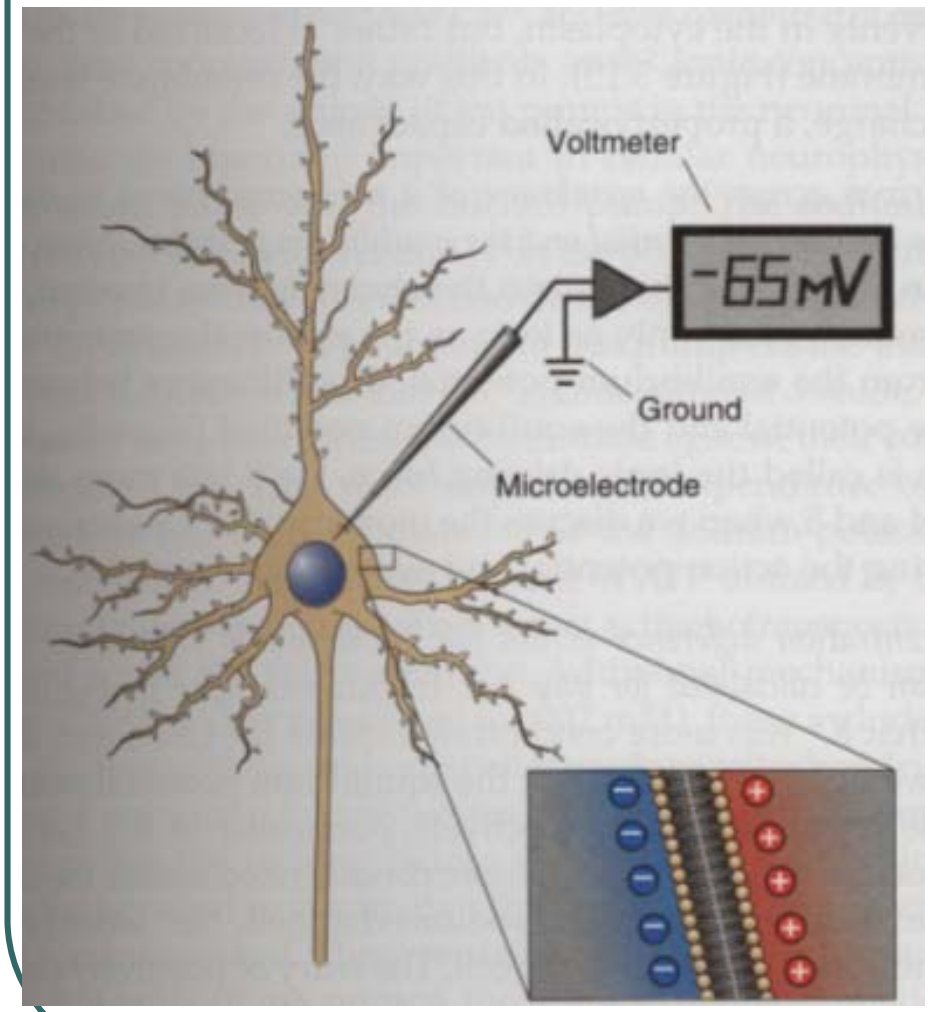
- About two-third of the neurons are cortical *pyramidal* cells.
  - The name, pyramidal, refers to the triangular shape of their cell bodies.
  - A typical pyramidal cell has a long *axon arising from* the base of the pyramidal and a long *apical dendrite* that *extends* toward the cortical surface.



The Pyramidal Cell - A Common Neuron



Membrane Potentials



- Membrane potential, V_m, is the potential difference between the inside and outside of the cell.
- Resting membrane potential of a typical neuron is about -65mV
  - which can be calculated using the Goldman equation.

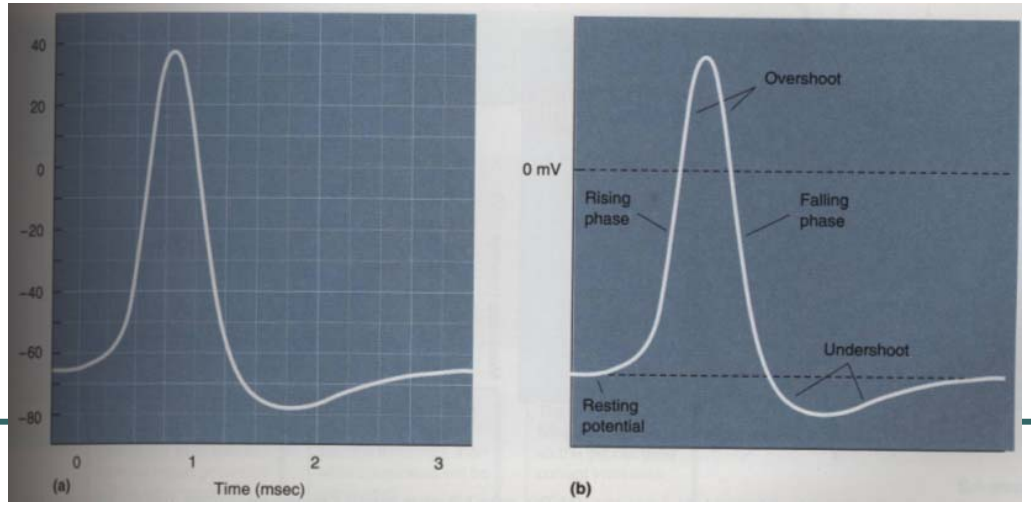
$$V_m = 61.54 \text{ mV} \log \frac{P_K [K^+]_o + P_{Na} [Na^+]_o}{P_K [K^+]_i + P_{Na} [Na^+]_i}$$

$$= 61.54 \text{ mV} \log \frac{40\ (5) + 1\ (150)}{40\ (100) + 1\ (15)}$$

$$= -65 \text{ mV}$$

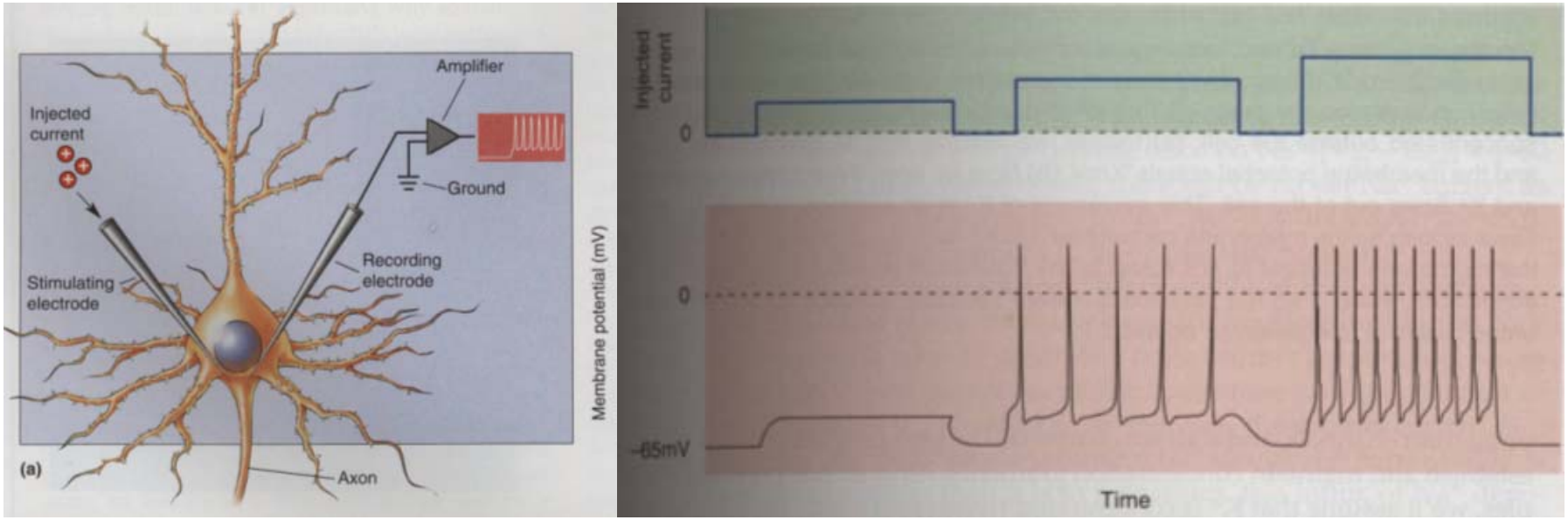
Action Potential

- 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

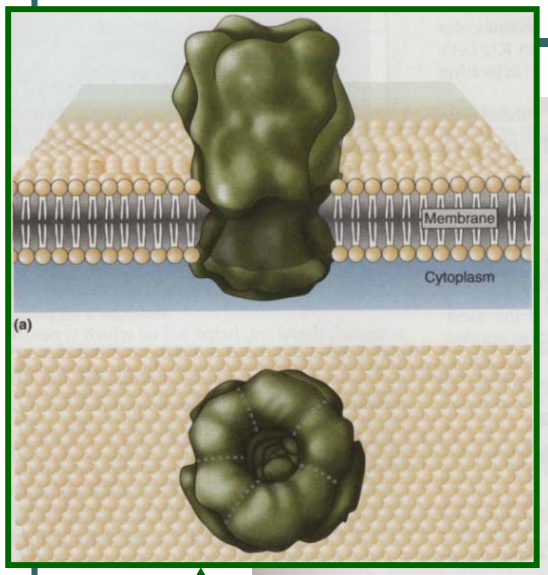


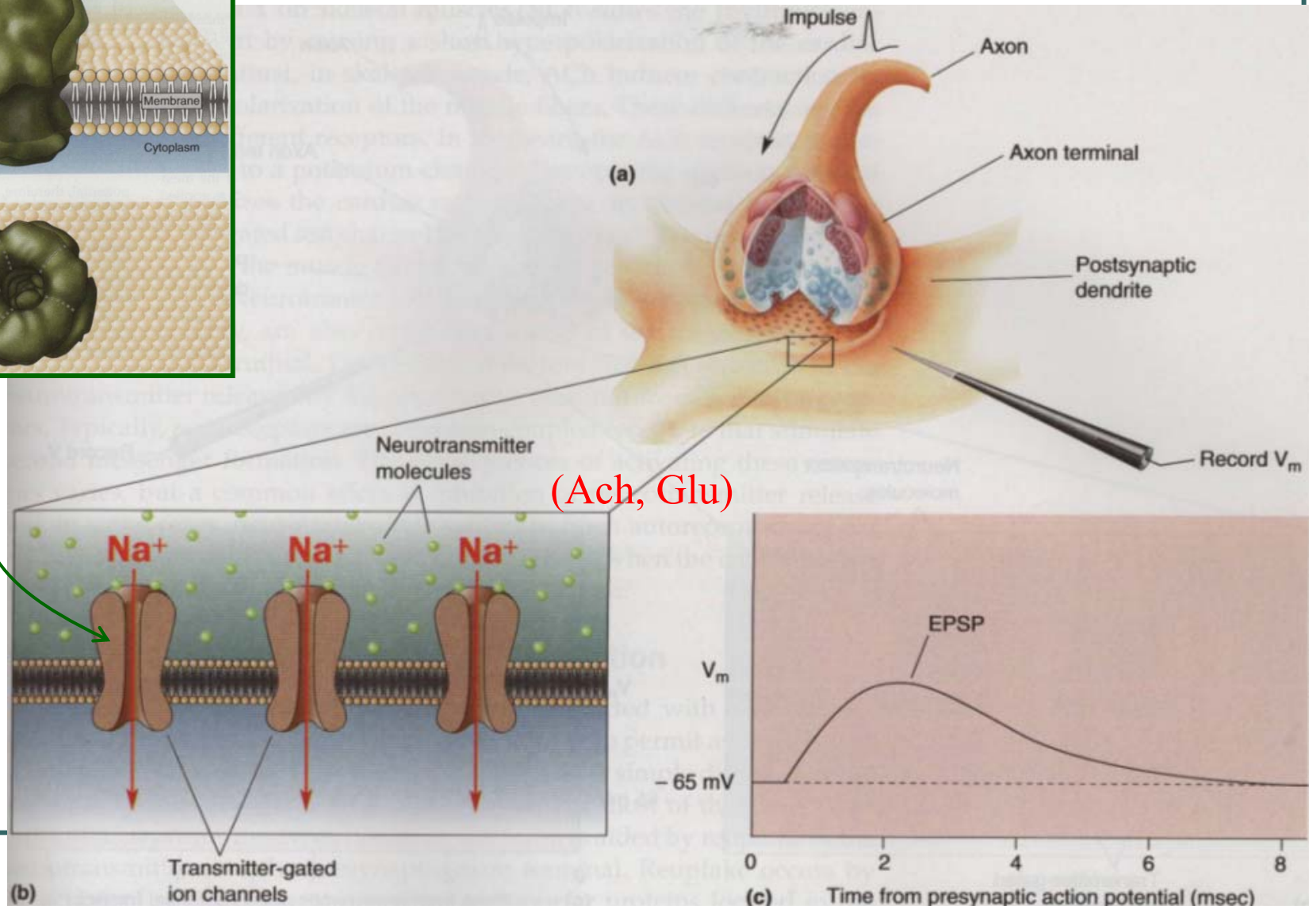
Generating Action Potentials

- Depolarization is usually caused by
  - Na^+ entry through channels sensitive to
    - Neurotransmitters
    - Electrical current

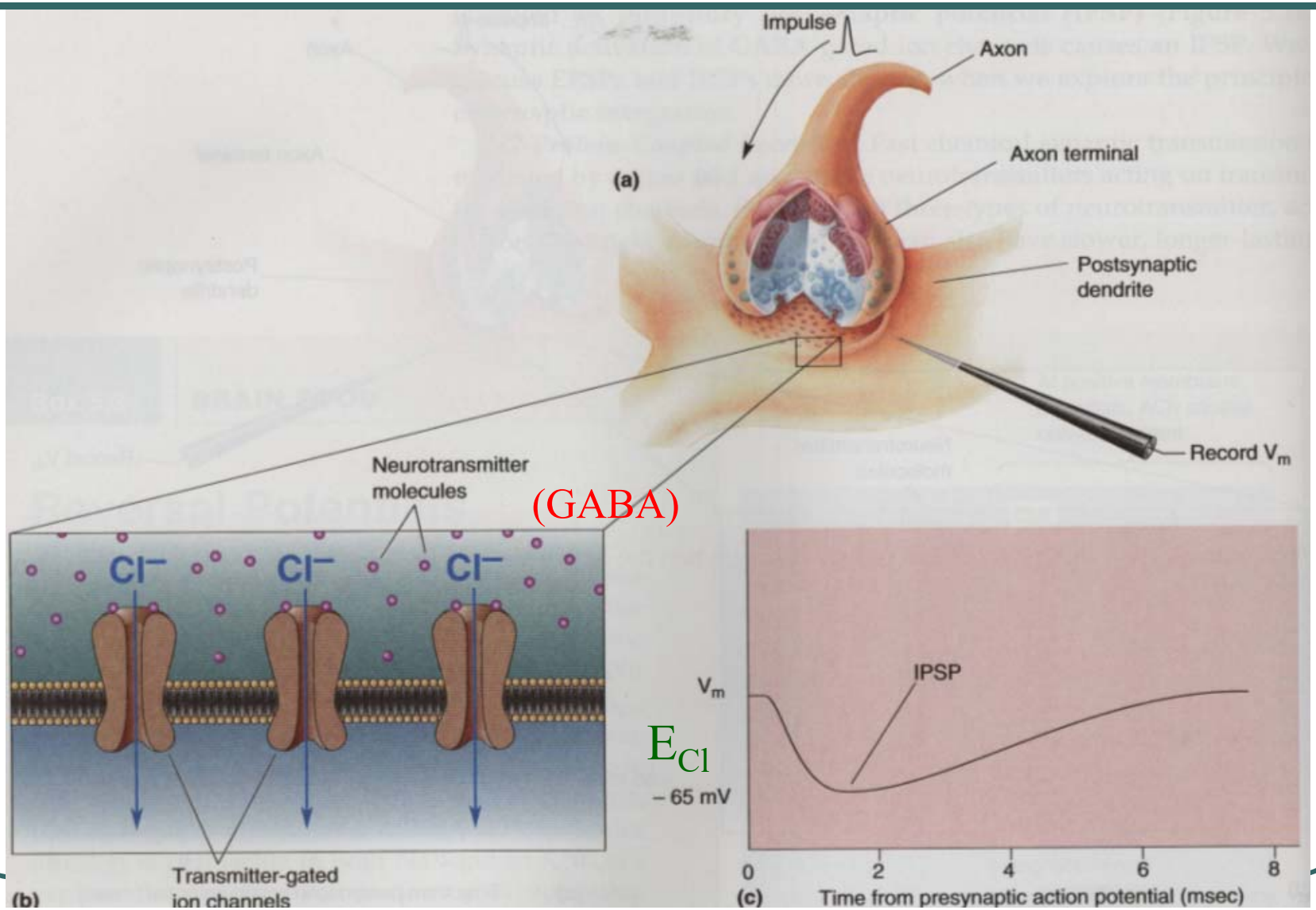


Excitatory Postsynaptic Potential (EPSP)

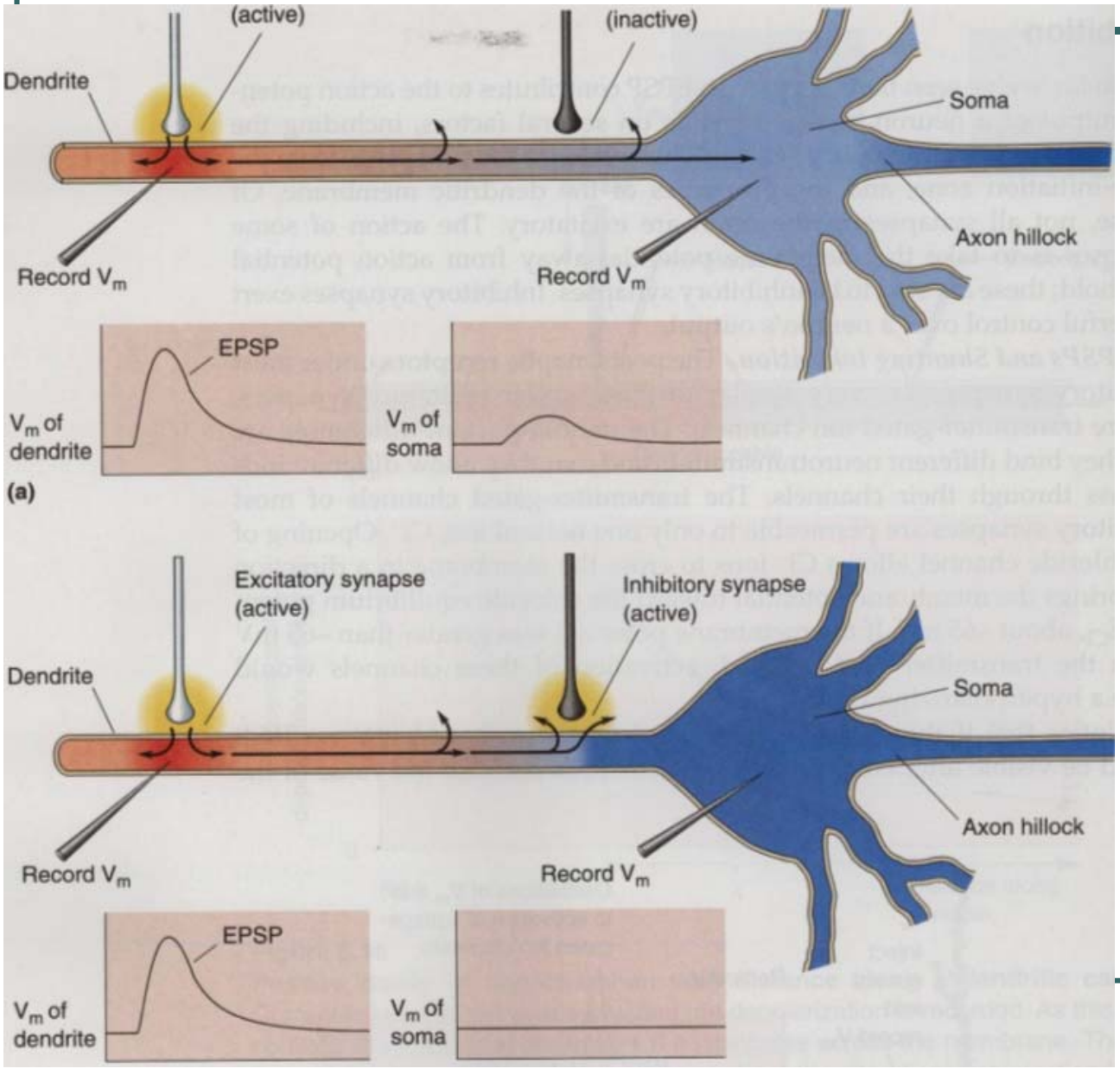




Inhibitory Postsynaptic Potential (IPSP)

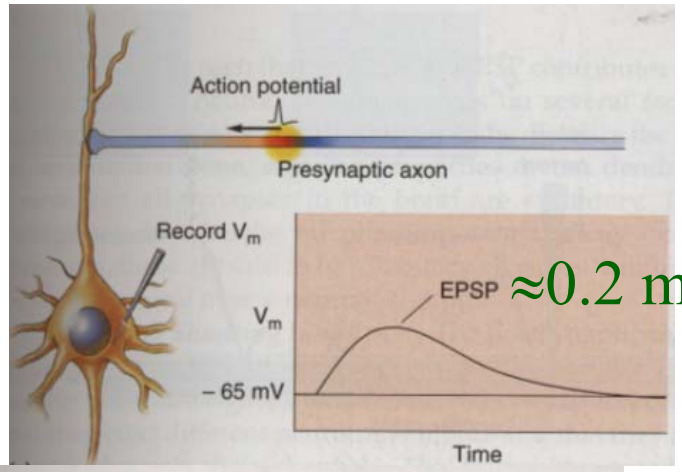


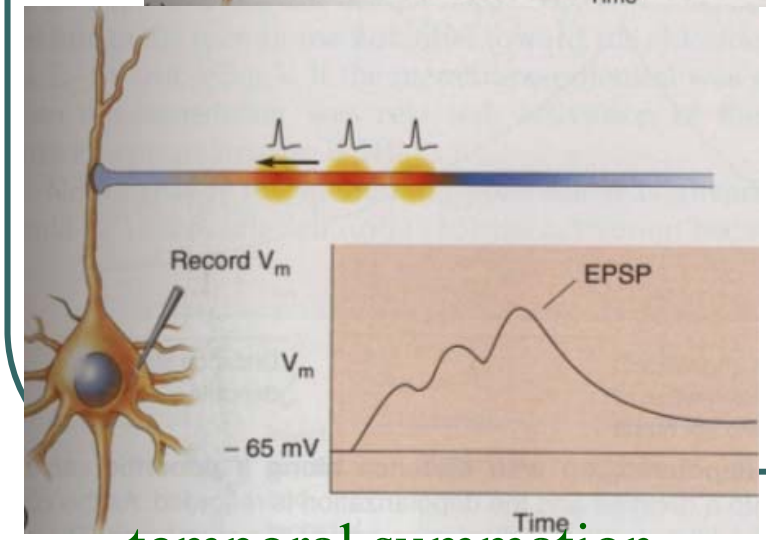
IPSP and Shunting Inhibition



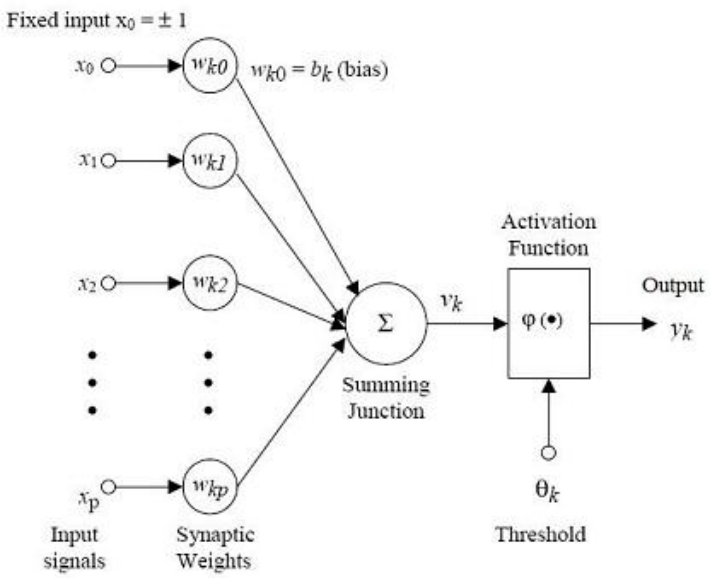
EPSP Integration

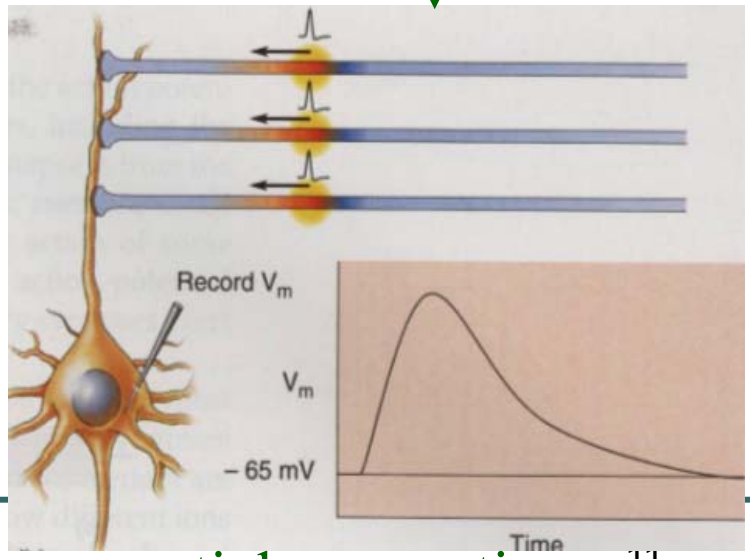
- Neurons in CNS perform computations.





temporal summation

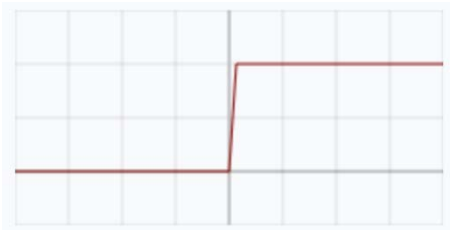




spatial summation

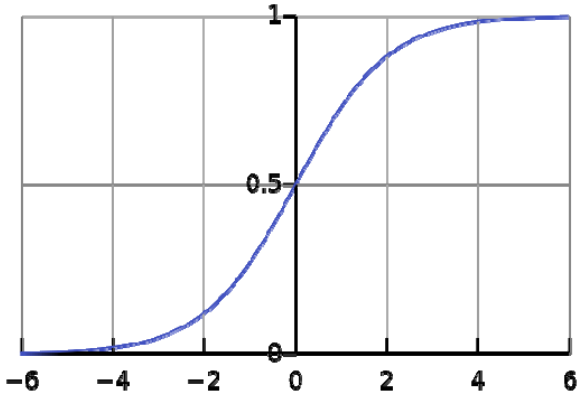
Activation Functions

- Nonlinearity of neural network
- Binary step function



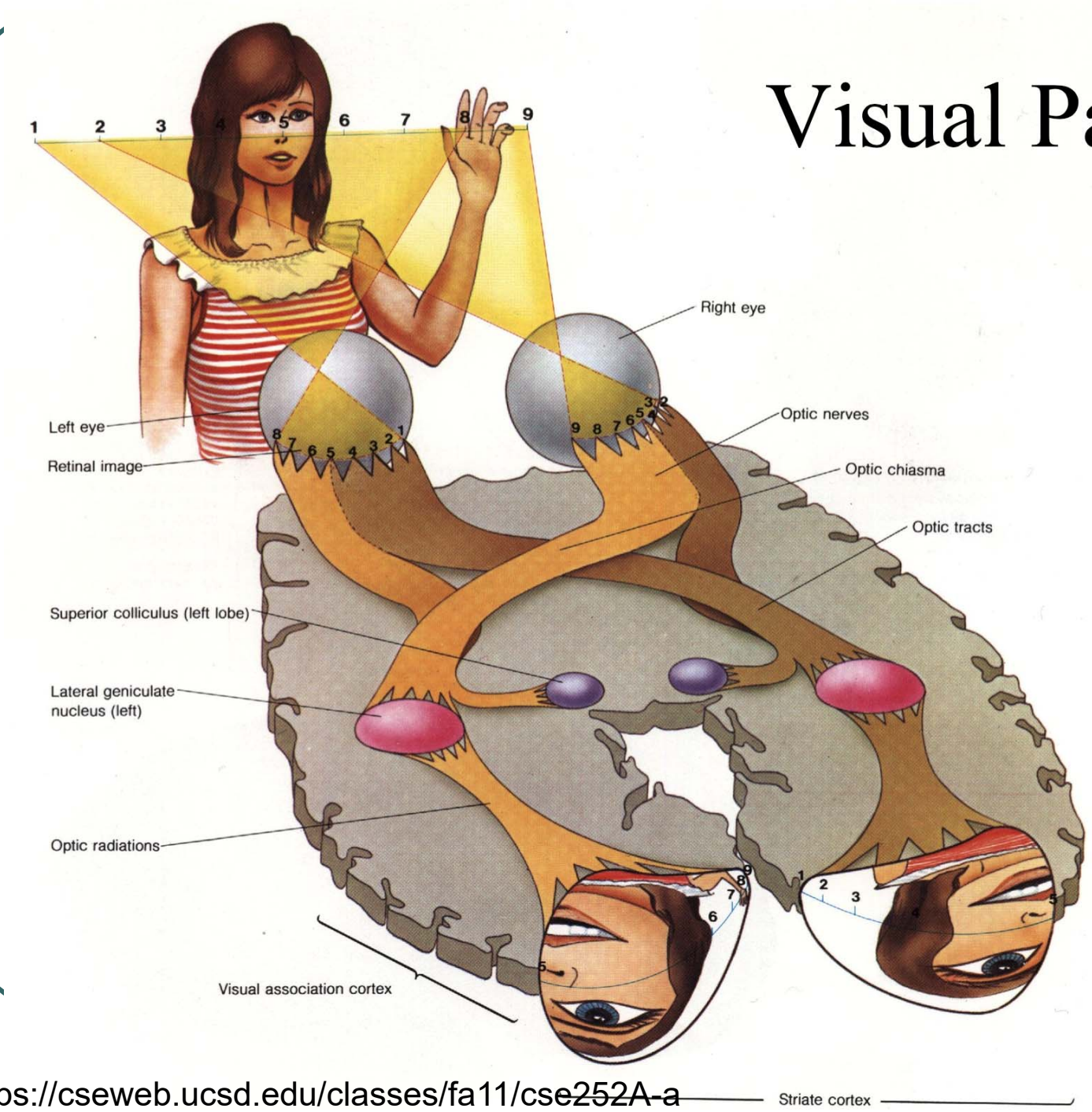
$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$$

- Sigmoid function



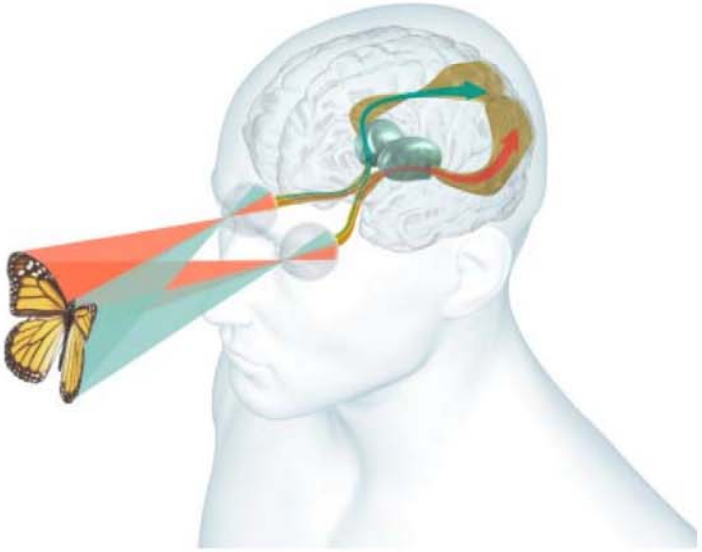
$$f(x) = \frac{1}{1 + e^{-x}}$$

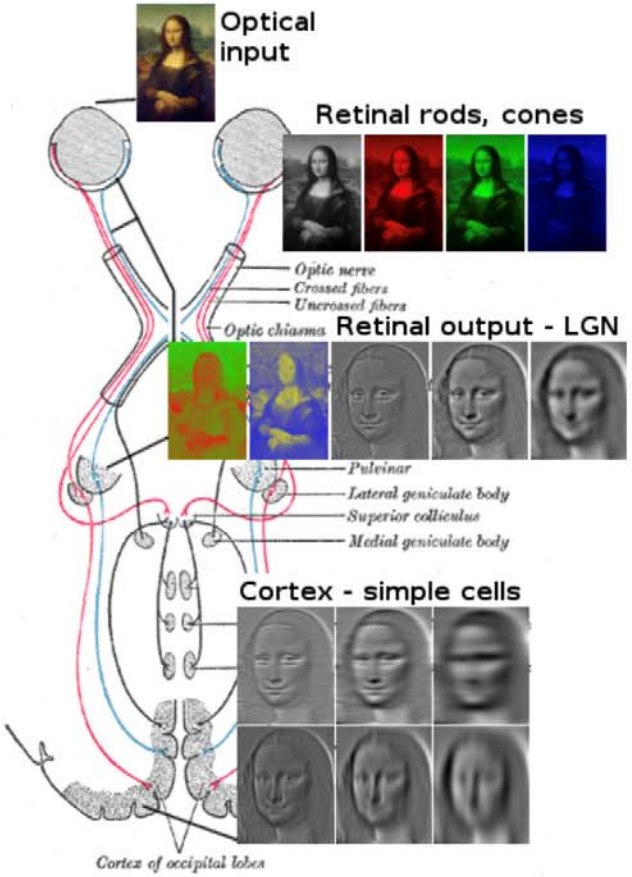
Visual Pathways

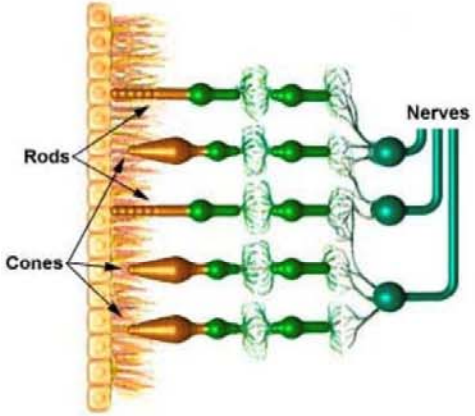


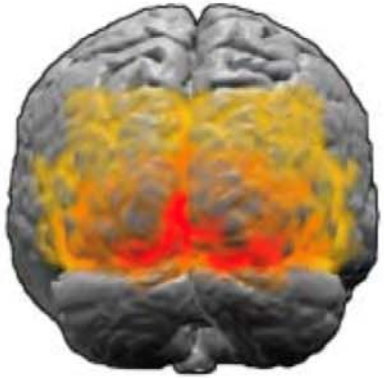
https://cseweb.ucsd.edu/classes/fa11/cse252A-a

Eye/Brain combination





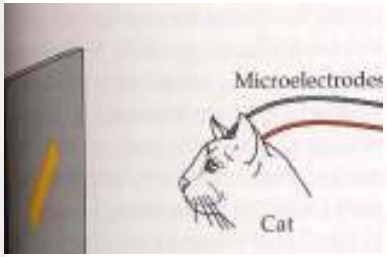




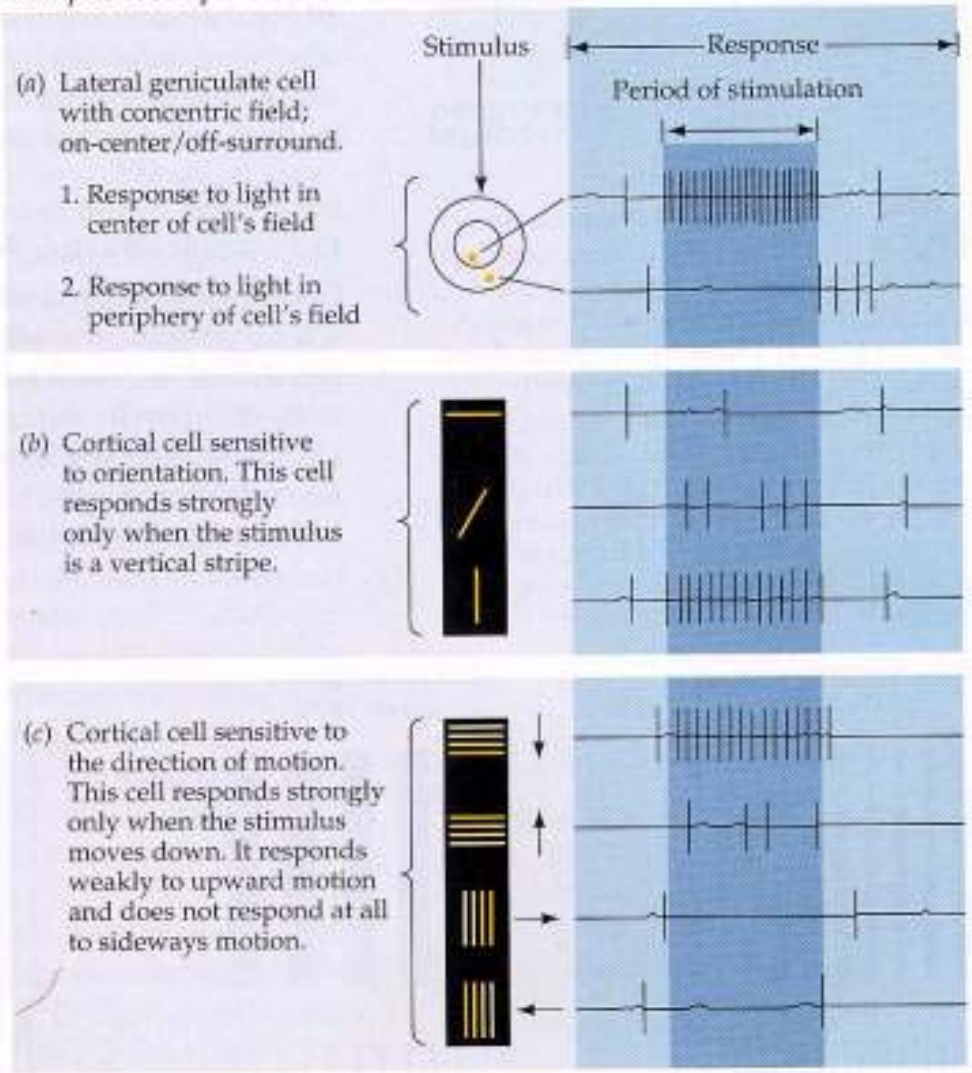
Receptive Fields of Lateral Geniculate and Primary Visual Cortex

David Hubel

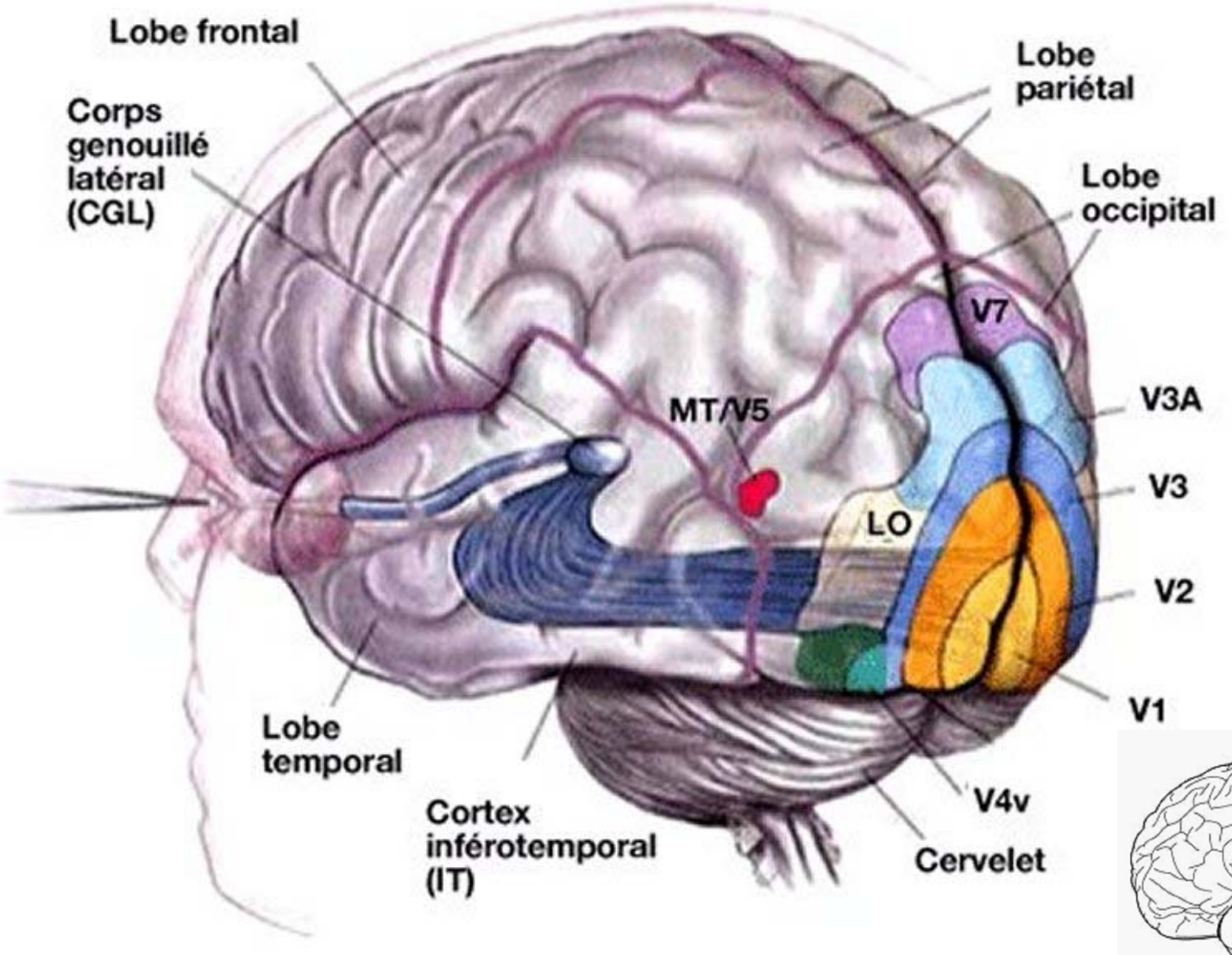
Torsten Wiesel

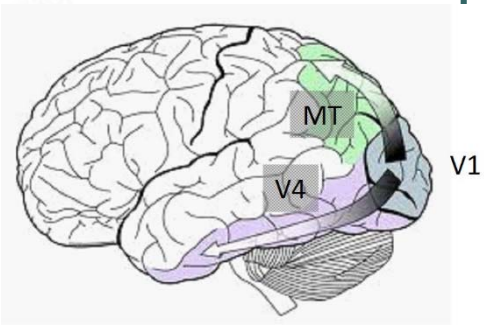


Examples of receptive fields of brain cells:



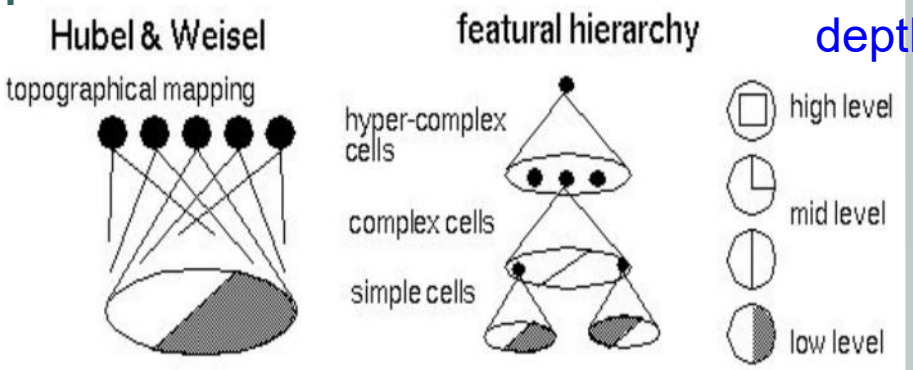
Human Cortical Visual Regions: V1, V2, V3, V4, V5 (MT)



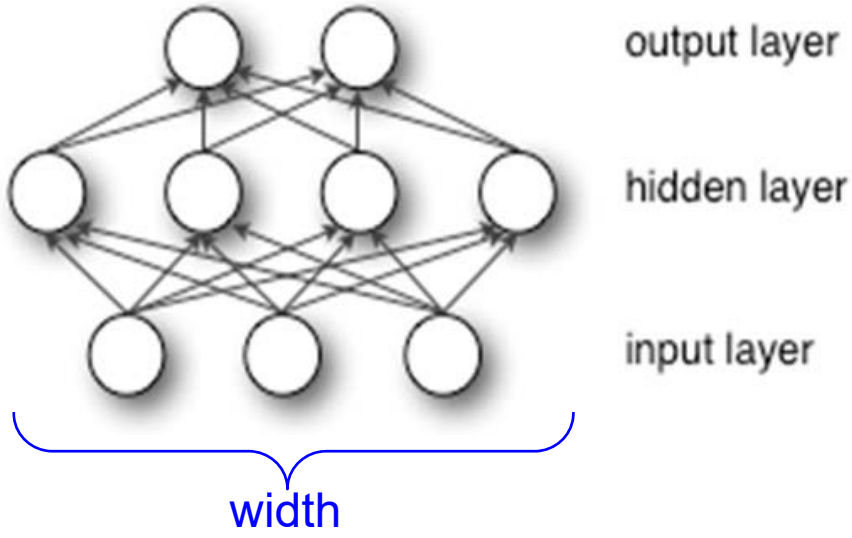


http://raymond.rodriguez1.free.fr/Documents/Organisme-A/Vision

Hubel/Wiesel Architecture and Multi-layer Perceptrons



Hubel and Weisel's architecture



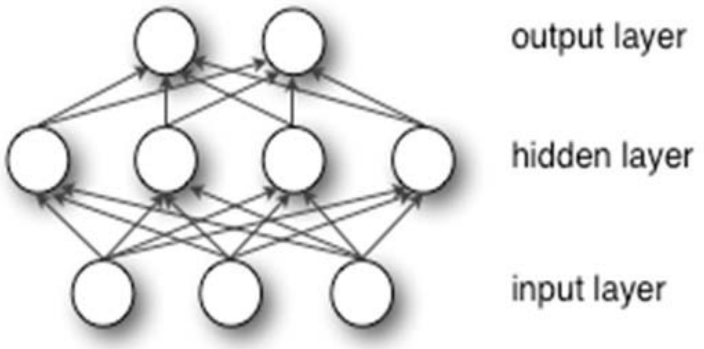
Multi-layer perceptrons
- A *non-linear* classifier

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 $f_w(x_i)$

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

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



From linear to nonlinear

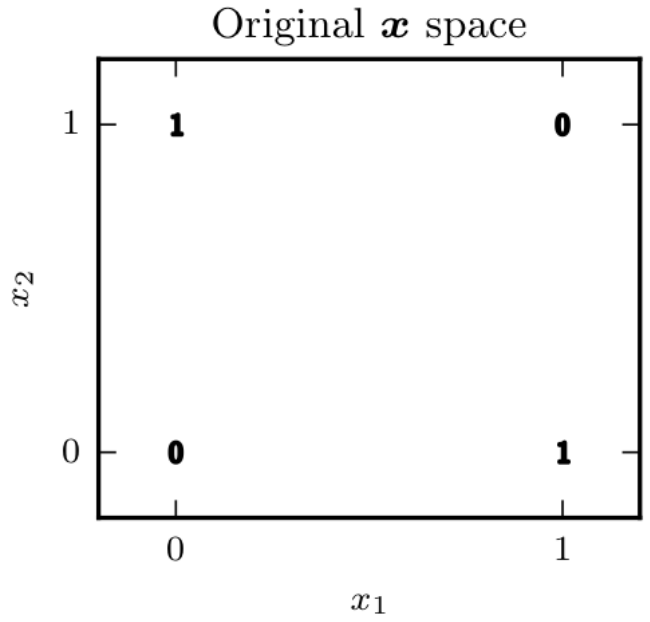
- To extend linear models to represent nonlinear function of ***x***: $y = \mathbf{w}^T\mathbf{x} + b \longrightarrow y = \mathbf{w}^T\phi(\mathbf{x}) + b$
  - To use kernel functions such as radial basis functions (RBFs), e.g.: $\phi(\mathbf{x}) = e^{-(\epsilon\|\mathbf{x}-\mathbf{x}_i\|)^2}$
  - Manually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

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

A simple example: learning XOR

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

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

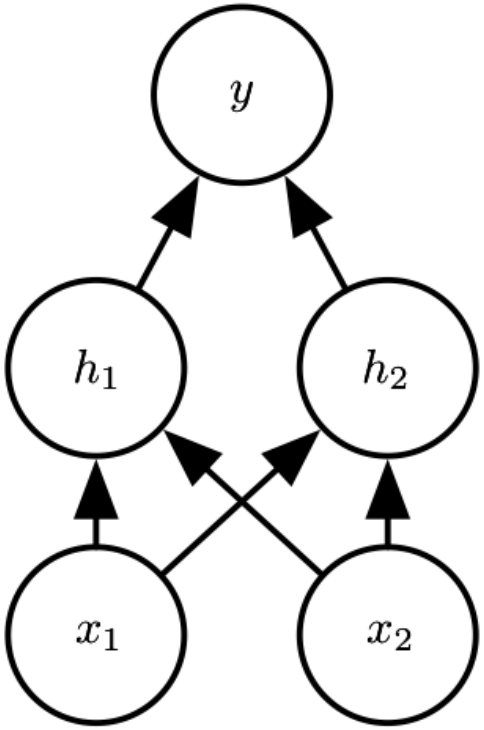


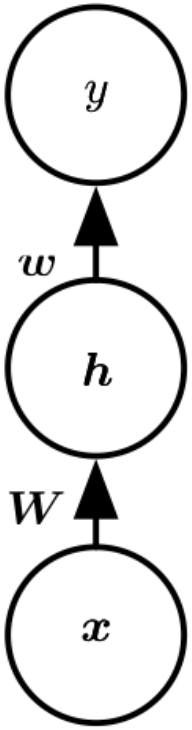
- Linear model is NOT able to represent XOR function.

A simple example: learning XOR

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

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





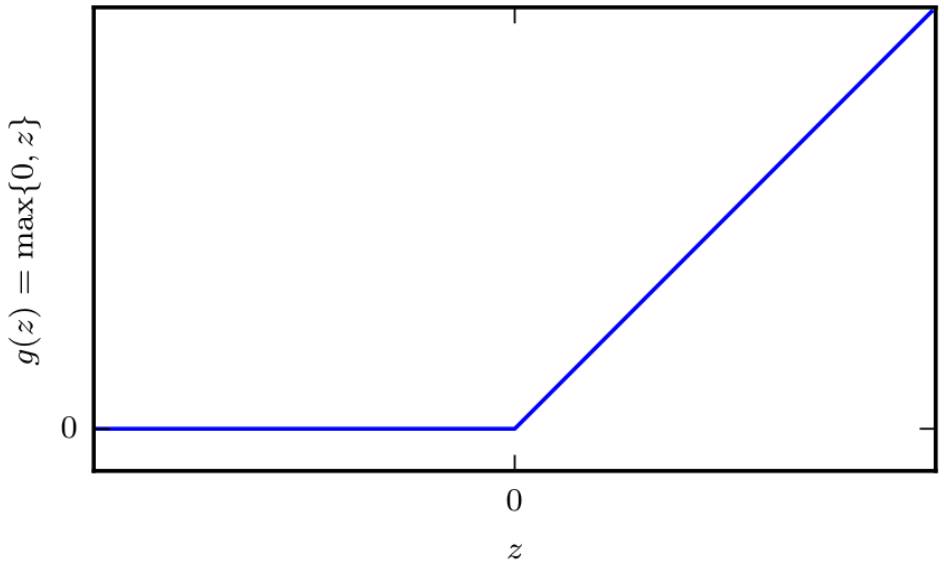
$$y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$$

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

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***:

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



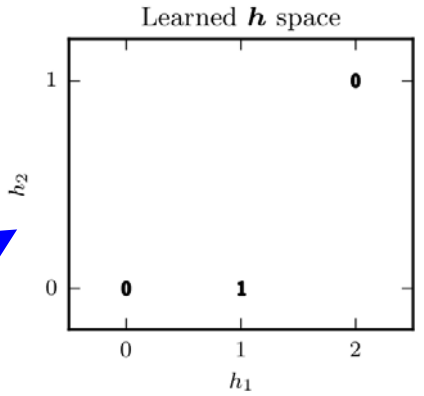
A simple example: learning XOR

- Complete neural network model:

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

- Obtain model parameters after training:

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



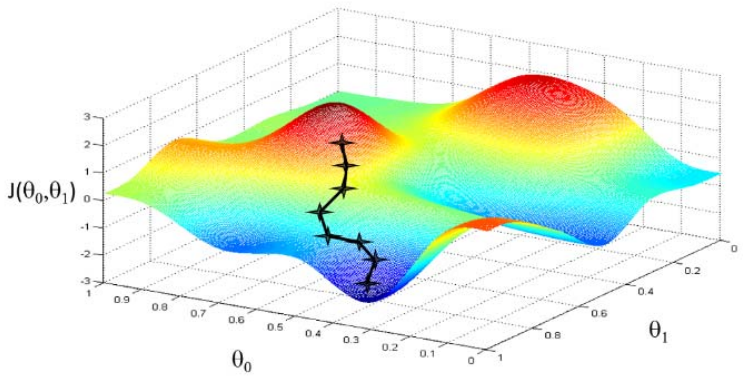
- Run the network:

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

$\mathbf{h}$

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 of small positive values
  - Issues:
    - Cost functions
    - Computation of gradients
    - Gradient-based optimization



Cost Functions: maximum likelihood

- Maximum likelihood model for the distribution of output *y*: $p(\mathbf{y}|\mathbf{x};\theta)$
- 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 $p_{\text{model}}(\mathbf{y}|\mathbf{x};\theta) = N(\mathbf{y}; f(\mathbf{x},\theta), \mathbf{I})$,

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

  - Large gradient and predictable for learning

Learning Conditional Statistics

- View the cost function as being a functional, rather than a function
  - A functional is a mapping from functions to real numbers
- Learning is to choose a function, not a set of parameters

$$f^* = \arg\min_f E_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|^2$$

$$f^*(\mathbf{x}) = E_{\mathbf{y}\sim p_{\text{data}}(\mathbf{y}|\mathbf{x})}[\mathbf{y}]$$

- Infinitely many samples give a function that predicts the mean of ***y*** for each value of ***x***.
- Combined with this mean squared error function, output units that saturate produce very small gradients.

Output Units

- The choice of cost function is tightly coupled with the choice of output unit.
- 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.

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].
  - Use a linear unit followed by thresholding:

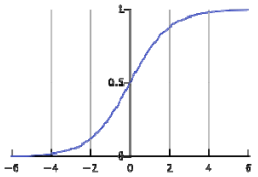
$$P(y = 1|\mathbf{x}) = \max\{0, \min\{1, \mathbf{w}^T\mathbf{h} + b\}\}$$

    - 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:

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



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

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

$$\log \tilde{P}(y) = yz$$

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

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

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

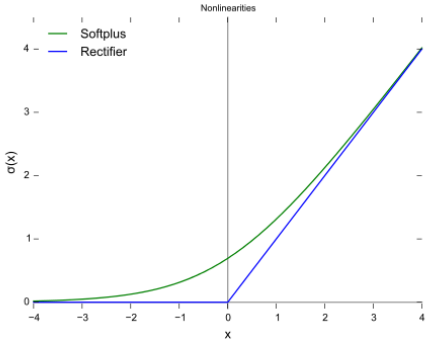
Output Units

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

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

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

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



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.

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

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

Output Units

- Softmax units for Multinoulli output distributions
  - Maximize 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 maximizing, the first term encourages z_i to be pushed up, while the second term encourages all of $\boldsymbol{z}$ to be pushed down.
  - Negative log-likelihood cost function always strongly penalizes the most active incorrect prediction.
  - Most objective function other than the log-likelihood do not work well with the softmax function.

Output Units

- Softmax units for Multinoulli output distributions
  - Softmax activation saturate when the differences between input values become extreme.
  - 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

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

Attention activates winner-take-all competition among visual filters

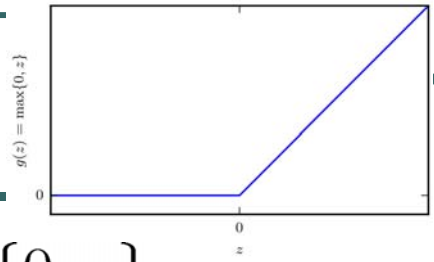
D.K. Lee[1,2], L. Itti[1,2], C. Koch[1] & J. Braun[1]

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



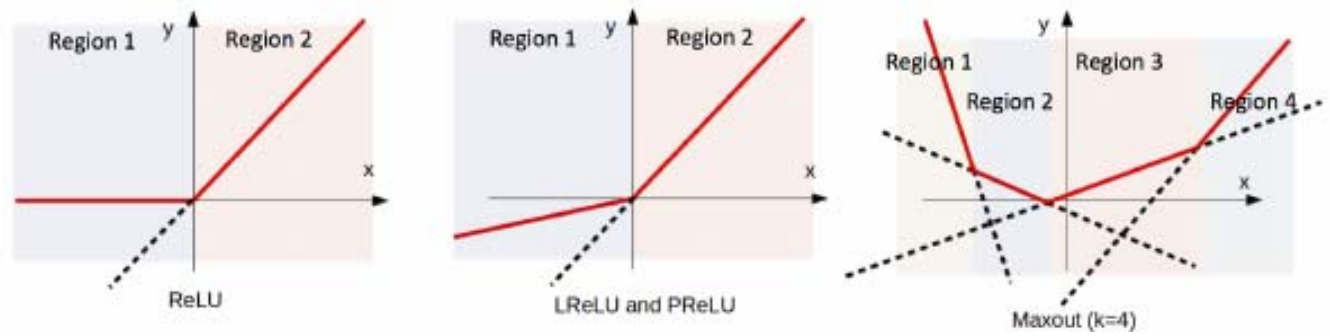
- ReLU: rectified linear unit: $g(z) = \max\{0, z\}$
- Easy to optimize because the derivatives through ReLU remain large whenever the unit is active.
- When initializing the parameters of the affine transformation, it can be a good practice to set all elements of ***b*** to a small, positive value, such as 0.1. $\mathbf{h} = g(\mathbf{W}^T\mathbf{x} + \mathbf{b})$
- 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 α_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.

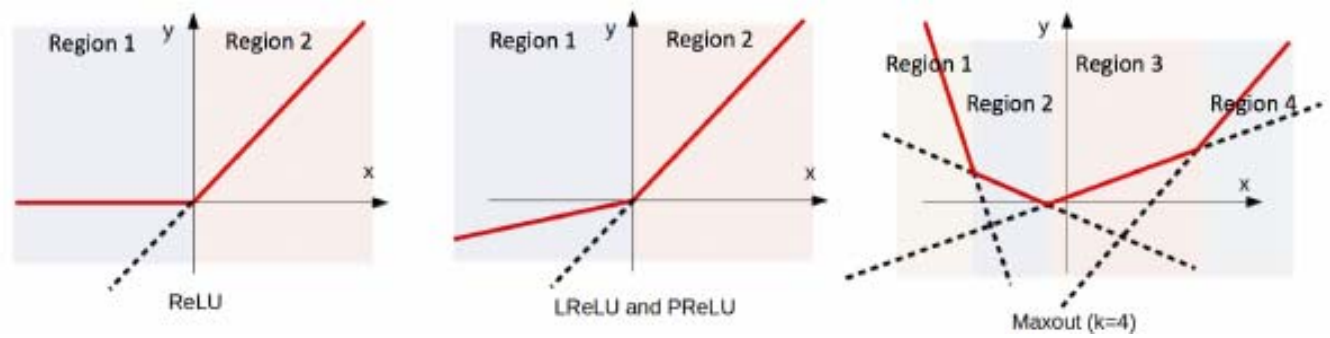


Generalizations of ReLUs

- Maxout units:

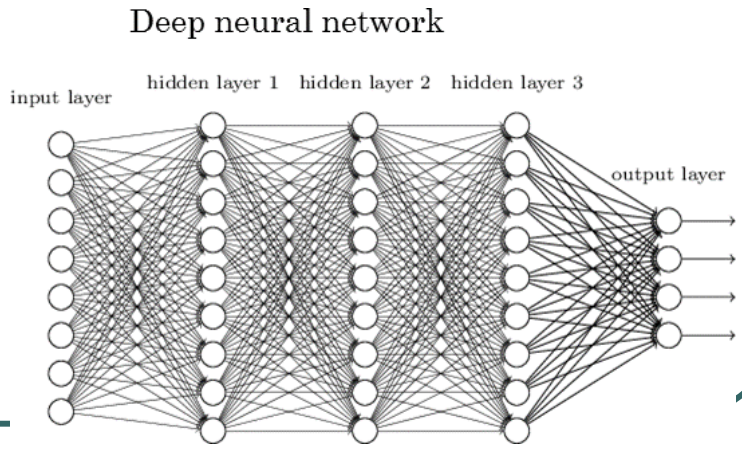
$$g(\mathbf{z})_i = \max_{j \in [1,k]} z_{ij} = \max\{\mathbf{w}_1^T\mathbf{x} + \mathbf{b}_1, \mathbf{w}_2^T\mathbf{x} + \mathbf{b}_2, \ldots, \mathbf{w}_k^T\mathbf{x} + \mathbf{b}_k\}$$

  - 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.



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

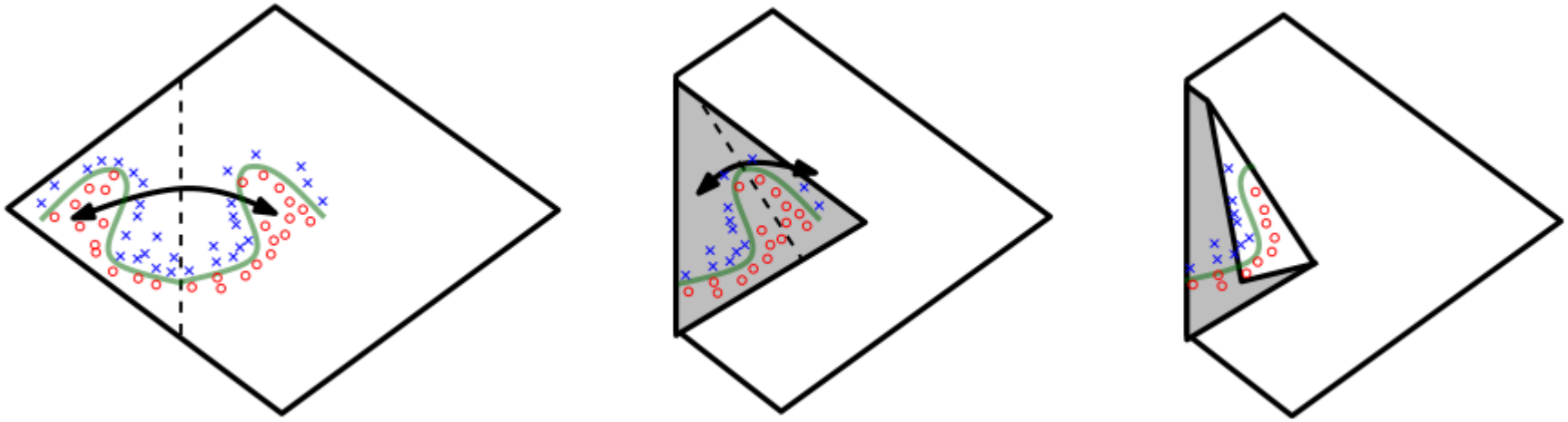


Universal Approximation Properties and Depth

- The universal approximation theorem states that a feedforward network with linear output layer and at least one hidden layer with any “squashing” activation function can approximate any Borel measurable function, provided that the network is given enough hidden units.
- A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to lean and generalize correctly.
- Using deeper models can reduce the number of units required to represent the desired function and can reduce the generalization error.

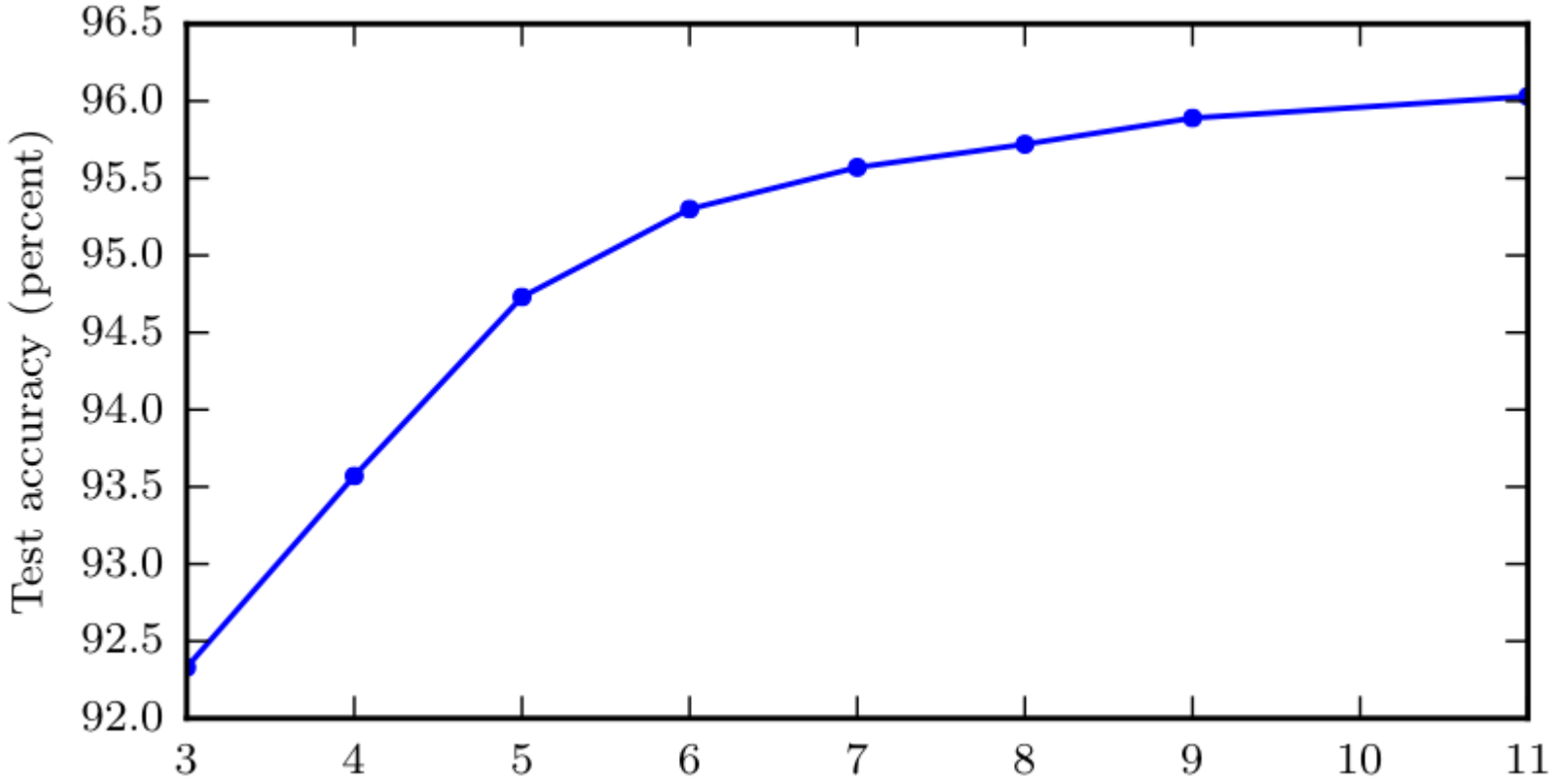
Universal Approximation Properties and Depth

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

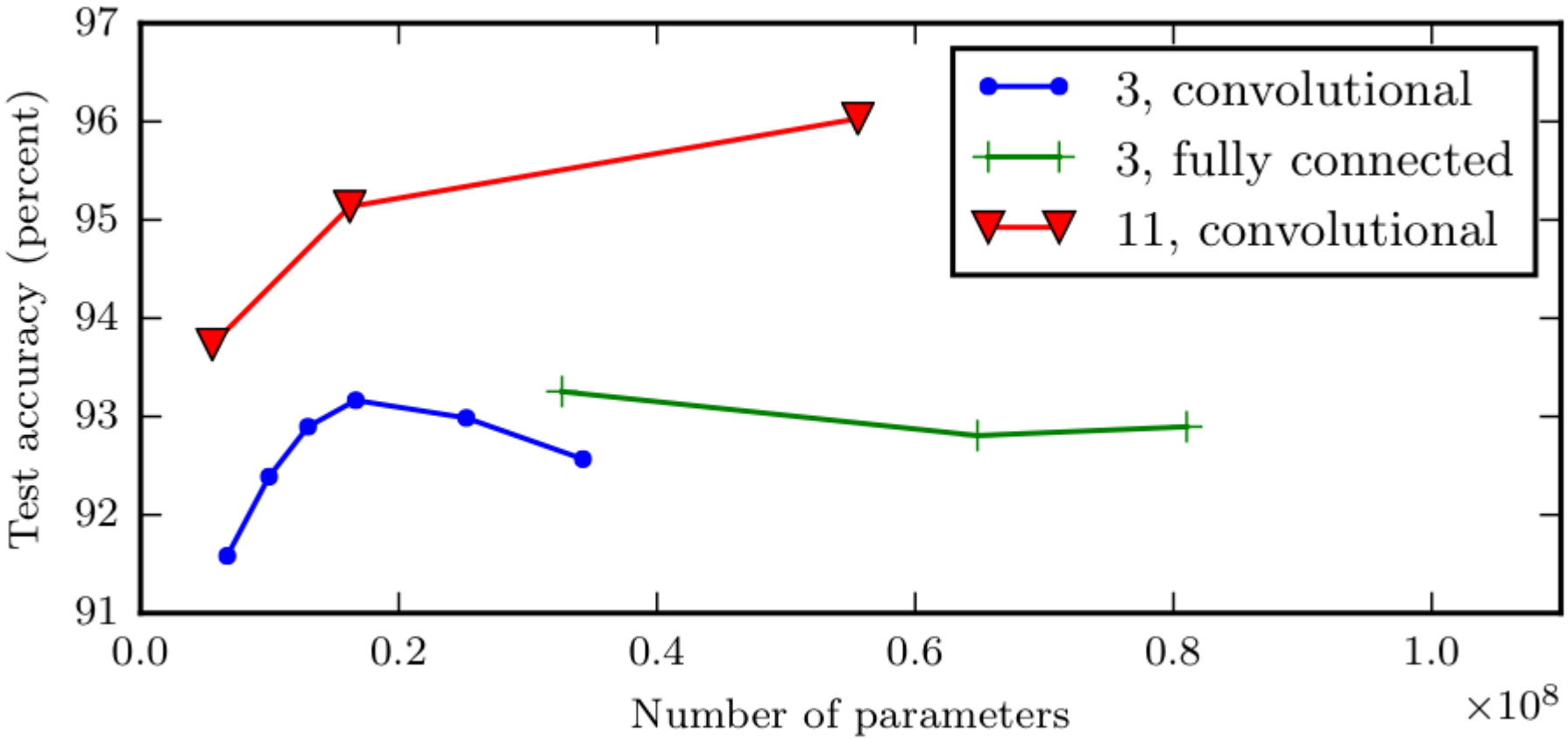


Better Generalization with Greater Depth

- SVHN

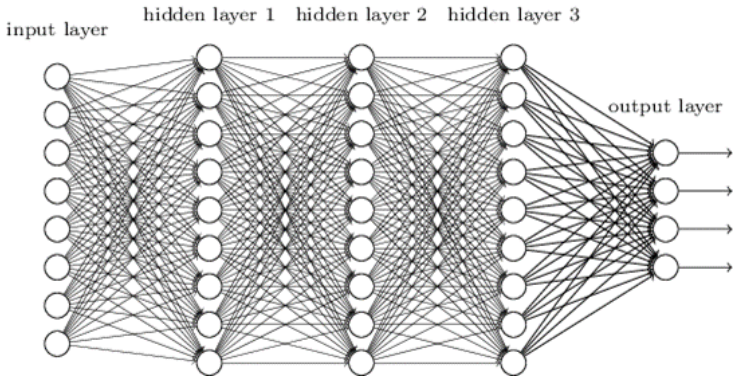


Large, Shallow Models Overfit More



Back-Propagation

- During inference: $\mathbf{x}$ → $\mathbf{y}$ (Deep neural network)

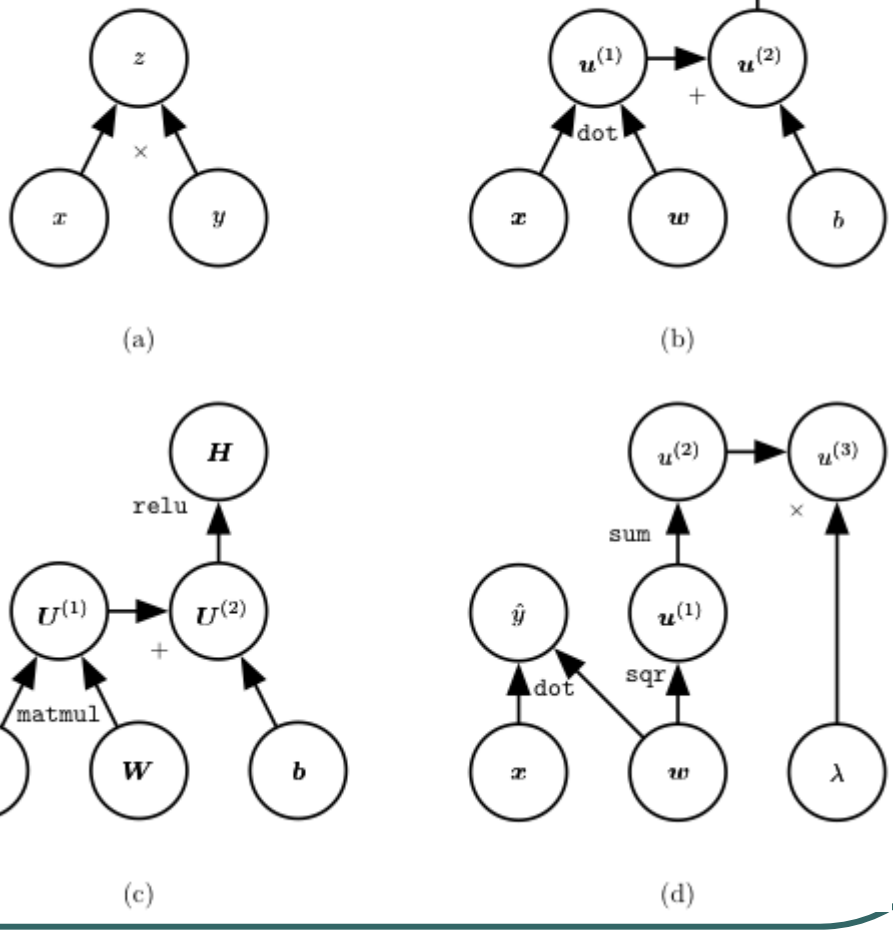


- During training: $\mathbf{x} \rightarrow J(\theta)$
- Backpropagation: $J(\theta) \rightarrow$ compute gradients $\nabla_\theta J(\theta)$
- Stochastic gradient descent is used to perform the learning using these gradients.

Computational Graphs

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

  (multi-dimensional array)



Chain Rule of Calculus

- Suppose $y = g(x), z = f(y) = f(g(x))$

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

- Suppose $\mathbf{x} \in R^m, \mathbf{y} = \mathbf{g}(\mathbf{x}) \in R^n, z = f(\mathbf{y}) = f(\mathbf{g}(\mathbf{x}))$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the *n*x*m* Jacobian matrix of ***g***

- Tensor $\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} \mathbf{Y}_j) \frac{\partial z}{\partial \mathbf{Y}_j}$

Backprop by Recursively Applying the Chain Rule

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

Backprop by Recursively Applying the Chain Rule

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation (Algorithm 6.1 for this example) to obtain the activations of the network

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[\partial u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return $\{$`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i\}$

Backprop in Fully-Connected MLP

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 Sec. 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 Sec. 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
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Backprop in Fully-Connected MLP

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:

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

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:

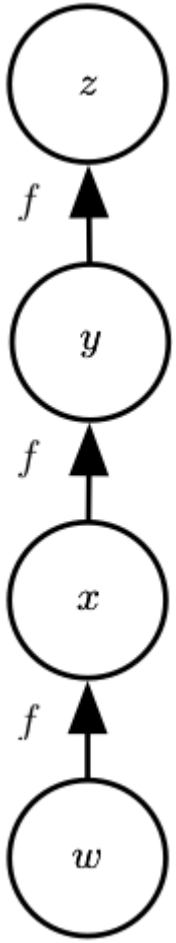
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

Repeated Subexpressions

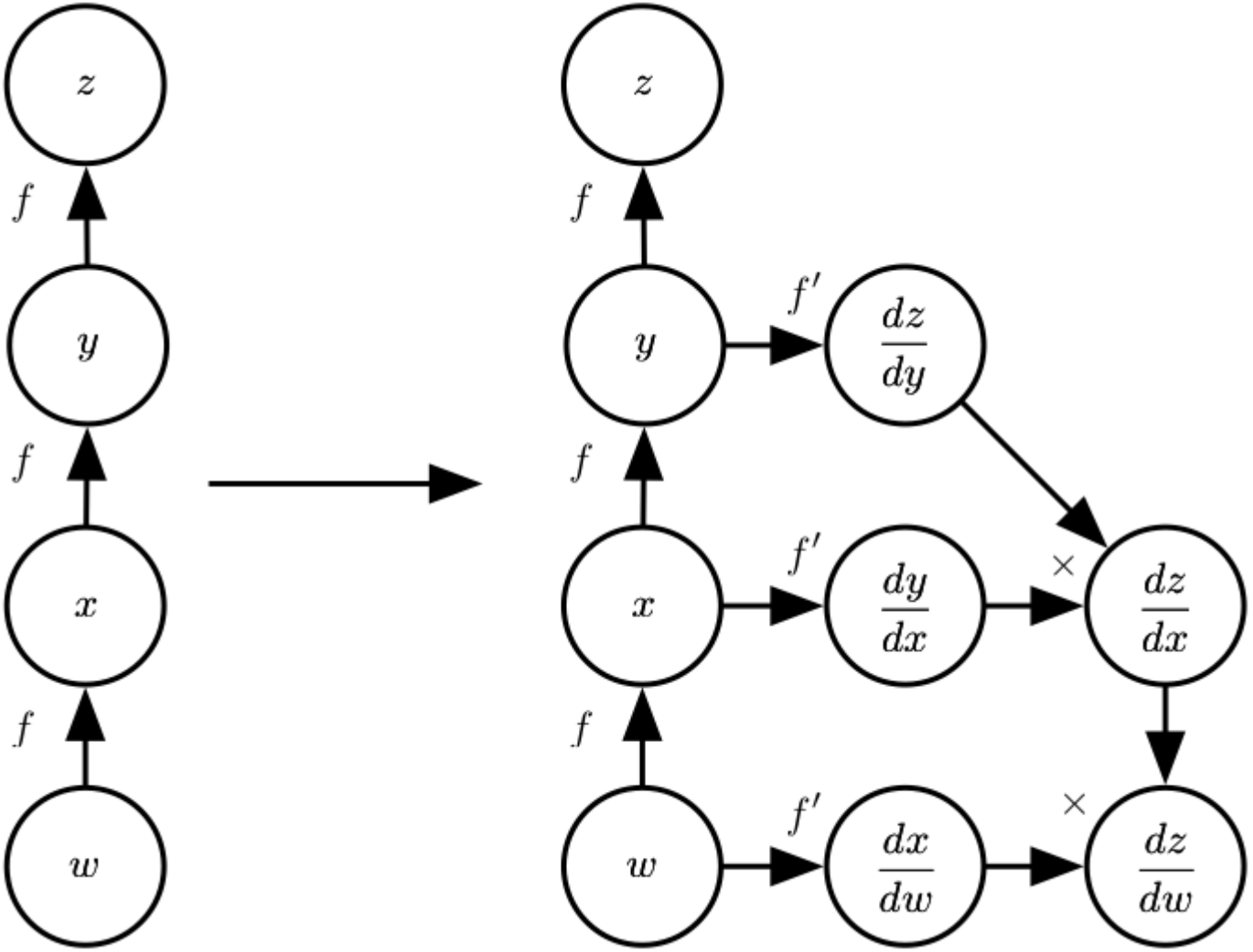
- Torch and Caffe

$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$



Symbol-to-Symbol Differentiation

- Theano and TensorFlow



General Backprop Algorithm

Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the `build_grad` subroutine of Algorithm 6.6

Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: z, the variable to be differentiated
Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of z and descendents of nodes in $\mathbb{T}$.
Initialize `grad_table`, a data structure associating tensors to their gradients
`grad_table`$[z] \leftarrow 1$
for $\mathbf{V}$ in $\mathbb{T}$ **do**
  `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$
end for
Return `grad_table` restricted to $\mathbb{T}$

General Backprop Algorithm

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`$(\texttt{get\_inputs}(\mathbf{C}, \mathcal{G}'), \mathbf{V}, \mathbf{D})$
    $i \leftarrow i + 1$
  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 $\mathbf{G}$

Backprop for MLP Training