Artificial Intelligence
CE-417, Group 1
Computer Eng. Department
Sharif University of Technology

Fall 2023

By Mohammad Hossein Rohban, Ph.D.

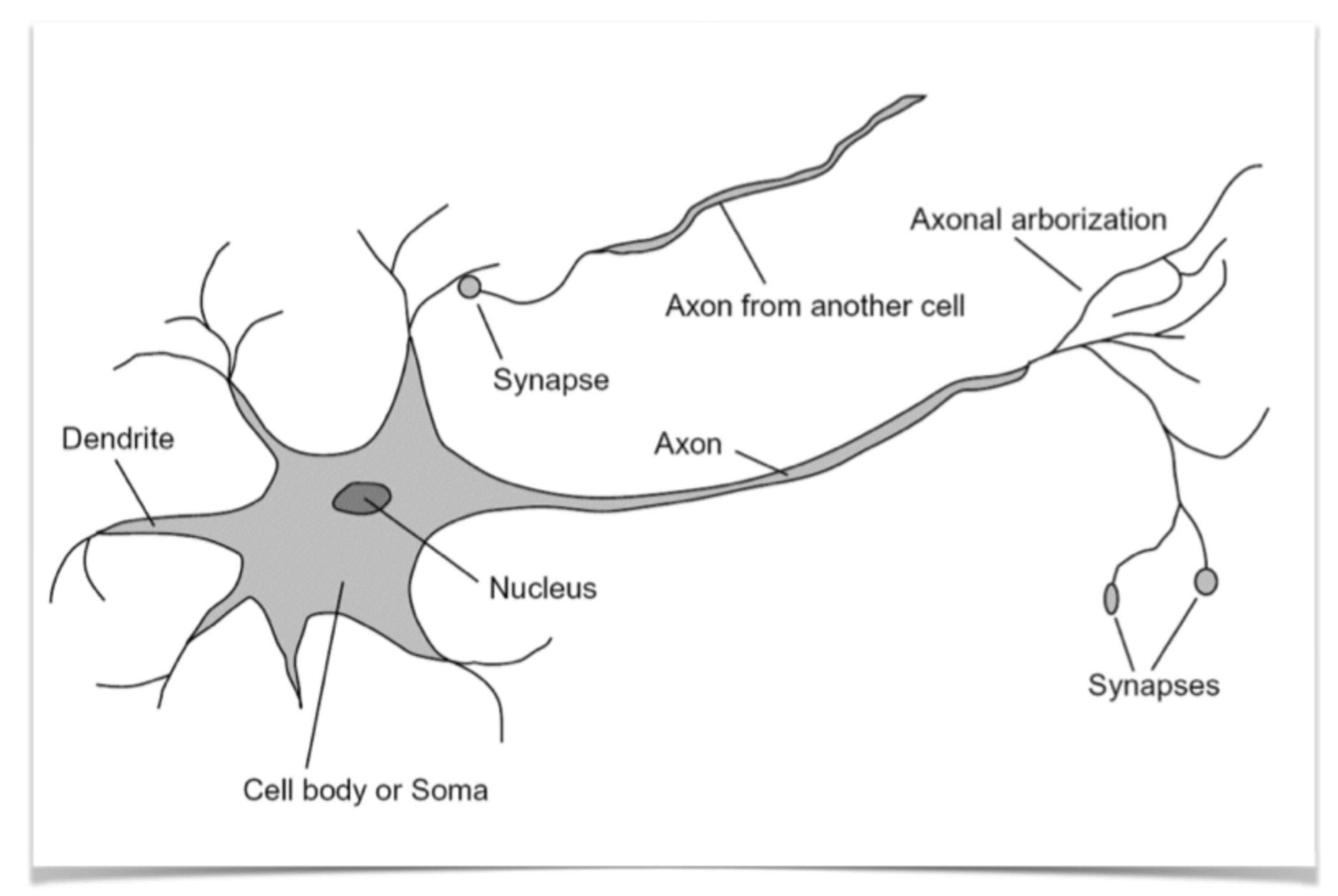
Courtesy: Most slides are adopted from CSE-573 (Washington U.), original slides for the textbook, and CS-188 (UC. Berkeley).

Neural Networks

Linear classifiers: Perceptron

- Decision trees
  - Inductive bias: use a combination of small number of features
- Nearest neighbor classifier (= **estimate the label as majority votes of k-NNs of input in the training data in the feature space**)
  - Inductive bias: all features are equally good
- Logistic Regression, and Perceptron
  - Inductive bias: use all features, but some more than others
    - learning weights for features

A neuron

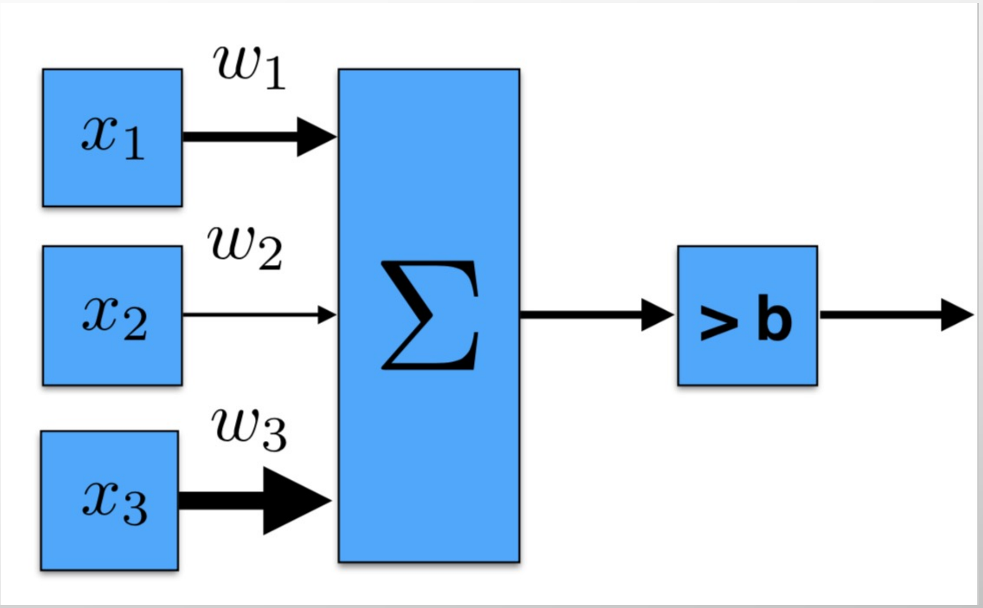

Axonal arborization
Axon from another cell
Synapse
Dendrite
Axon
Nucleus
Synapses
Cell body or Soma

Perceptron

- Input are feature values
- Each feature has a weight
- Sum in the activation
- If the activation is:
  - > b, output class 1
  - otherwise, output class 2

$$\text{activation}(\mathbf{w}, \mathbf{x}) = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

$$\mathbf{x} \rightarrow (\mathbf{x}, 1)$$
$$\mathbf{w}^T \mathbf{x} + b \rightarrow (\mathbf{w}, b)^T (\mathbf{x}, 1)$$



Example: spam

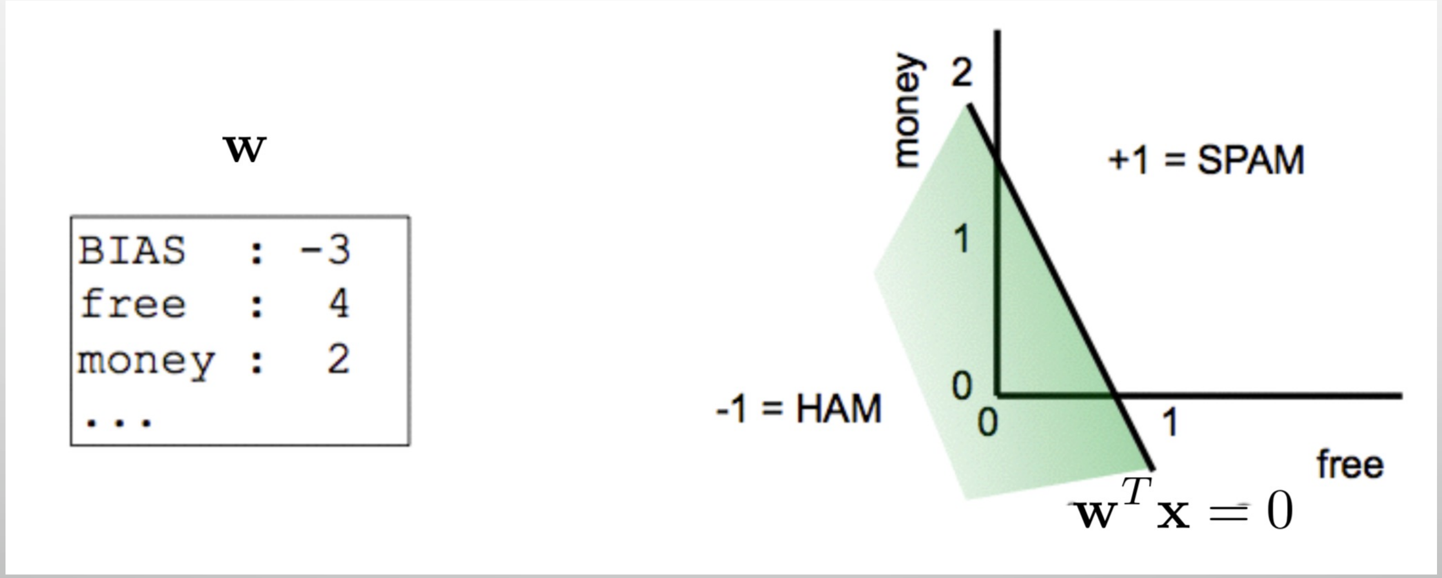
- Imagine 3 features (spam is “positive” class):
  - Free (number of occurrences of “free”)
  - Money (number of occurrences of “money”)
  - BIAS (intercept, always has value 1)

email	$\mathbf{x}$	$\mathbf{w}$	$\mathbf{w}^T\mathbf{x}$
“free money”	BIAS : 1	BIAS : -3	$(1)(-3)\ +$
	free : 1	free : 4	$(1)(4)\ +$
	money : 1	money : 2	$(1)(2)\ +$
	...	...	$\ldots$
			$= 3$

$$\mathbf{w}^T\mathbf{x} > 0 \rightarrow \text{SPAM!!}$$

Geometry of the perceptron

- In the space of feature vectors
  - Examples are points (in D dimensions)
  - A weight vector is a hyperplane (a D-1 dimensional object)
  - One side corresponds to y=+1
  - Other side corresponds to y=-1
- Perceptrons are also called as linear classifiers



Learning a perceptron

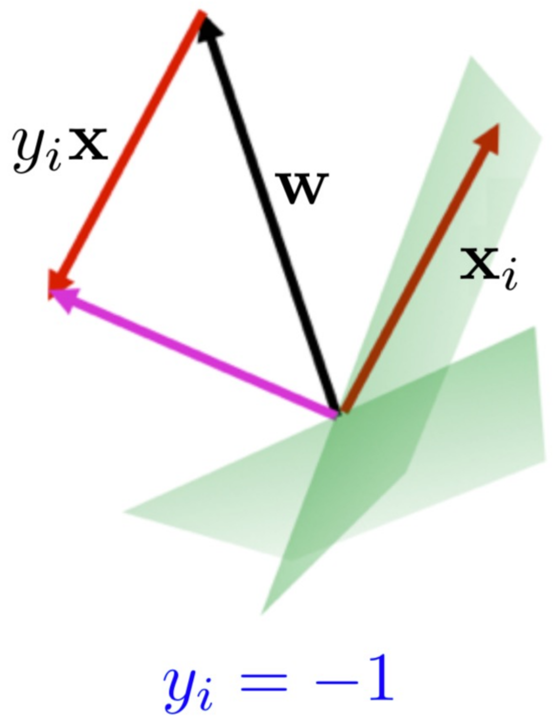
Input: training data $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$

Perceptron training algorithm [Rosenblatt 57]

- Initialize $\mathbf{w} \leftarrow [0, \ldots, 0]$
- for iter = 1,…,T
  - for i = 1,..,n
    - predict according to the current model

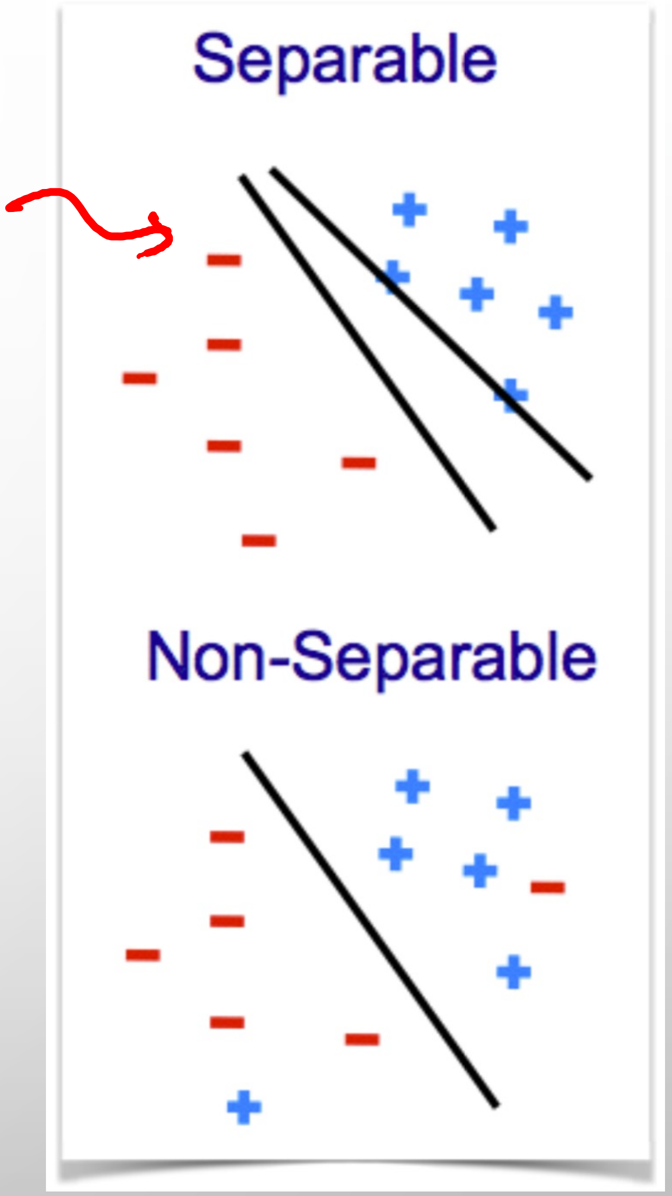
$$\hat{y}_i = \begin{cases} +1 & \text{if } \mathbf{w}^T \mathbf{x}_i > 0 \\ -1 & \text{if } \mathbf{w}^T \mathbf{x}_i \leq 0 \end{cases}$$

    - if $y_i = \hat{y}_i$, no change
    - else, $\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$



Properties of perceptrons

- **Convergence:** if the training data is separable then the perceptron training will eventually converge [block 62, novikoff 62]



Maximum margin

- Assuming that the data is linearly separable, we define margin as

$$\delta = \max_{\mathbf{w}} \min_{(\mathbf{x}_i, y_i)} \left[y_i \mathbf{w}^T \mathbf{x}_i \right]$$
$$\text{such that, } ||\mathbf{w}|| = 1$$

Proof of convergence

Assumption : $\|\hat{w}\| = 1$

Let, $\hat{\mathbf{w}}$ be the separating hyperplane with margin δ

w is getting closer

$$
\begin{aligned}
\hat{\mathbf{w}}^T \mathbf{w}^{(k)} &= \hat{\mathbf{w}}^T \left(\mathbf{w}^{(k-1)} + y_i \mathbf{x}_i \right) && \text{update rule} \\
&= \hat{\mathbf{w}}^T \mathbf{w}^{(k-1)} + \hat{\mathbf{w}}^T y_i \mathbf{x}_i && \text{algebra} \\
&\geq \hat{\mathbf{w}}^T \mathbf{w}^{(k-1)} + \delta && \text{definition of margin} \\
&\geq k\delta && ||\mathbf{w}^{(k)}|| \geq k\delta
\end{aligned}
$$

Proof of convergence (cont.)

bound the norm

$$
\begin{aligned}
||\mathbf{w}^{(k)}||^2 &= ||\mathbf{w}^{(k-1)} + y_i \mathbf{x}_i||^2 && \text{update rule} \\
&\leq ||\mathbf{w}^{(k-1)}||^2 + ||y_i \mathbf{x}_i||^2 && y_i \mathrm{w}^{(k-1)T} \mathrm{x}_i < 0 \\
&\leq ||\mathbf{w}^{(k-1)}||^2 + 1 && \text{norm} \\
&\leq k && ||\mathbf{w}^{(k)}|| \leq \sqrt{k}
\end{aligned}
$$

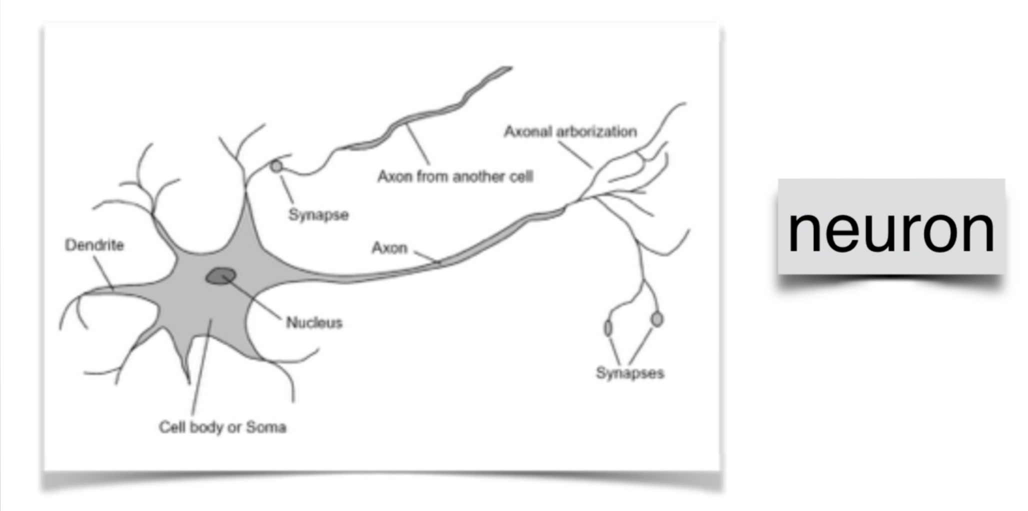
$$k\delta \leq ||\mathbf{w}^{(k)}|| \leq \sqrt{k} \longrightarrow k \leq \frac{1}{\delta^2}$$

Limitations of perceptrons

- **Convergence:** if the data isn't separable, the training algorithm may not terminate.
- **Overtraining:** test/validation accuracy rises and then falls.

Multi-Layered Perceptron : Motivation

- One of the main weakness of linear models is that they are linear.
- Decision trees and k-NN classifiers can model non-linear boundaries.
- Multi-layer neural networks are yet another non-linear classifier.
- Take the biological inspiration further by chaining together perceptrons.
- Allows us to use what we learned about linear models:
  - Loss functions, regularization, optimization



Two-layer network architecture

$$y = \mathbf{v}^T \mathbf{h}$$

Non-linearity is important

link function

$$h_i = f(\mathbf{w}_i^T \mathbf{x})$$

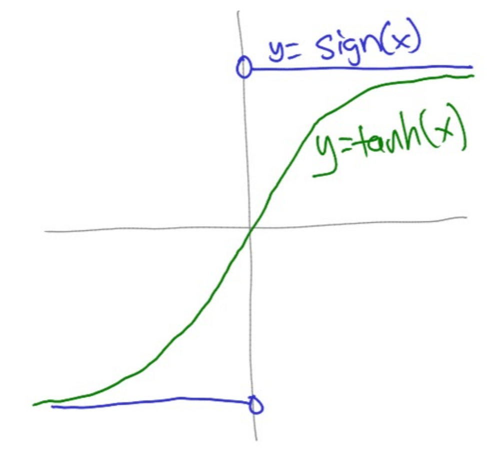
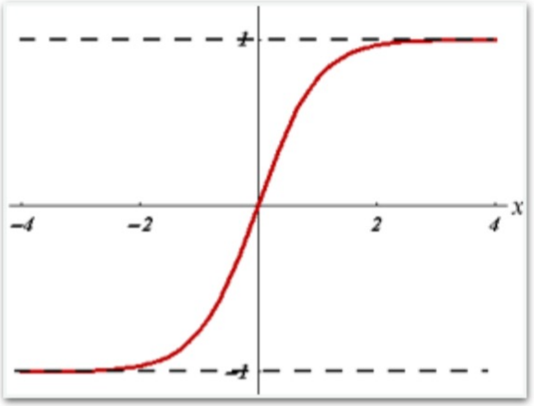
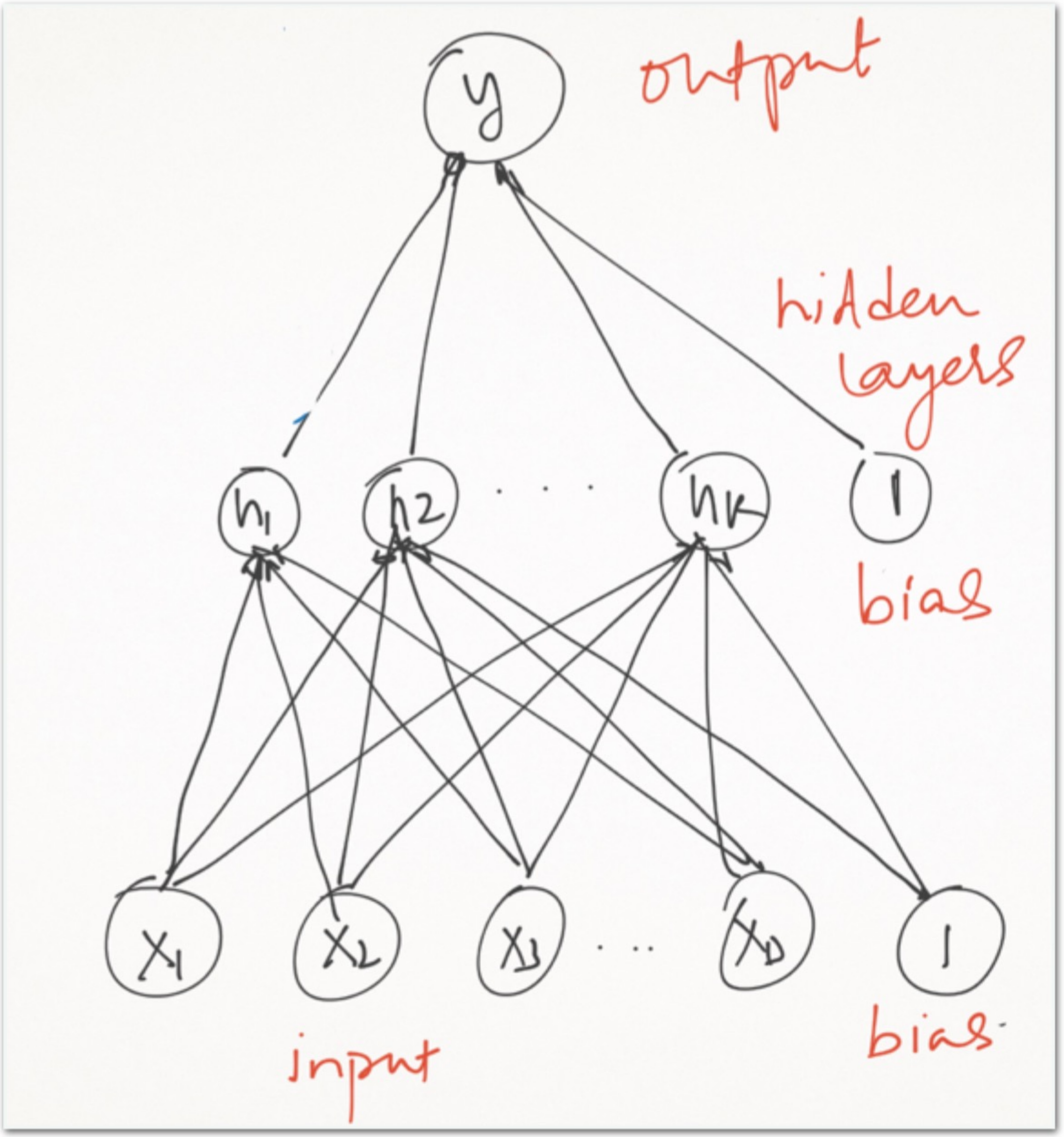


Figure 10.2: picture of sign versus tanh
[1] It's derivative is just $1 - \tanh^2(x)$.

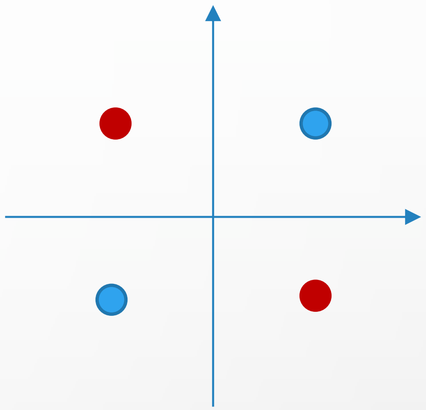


$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

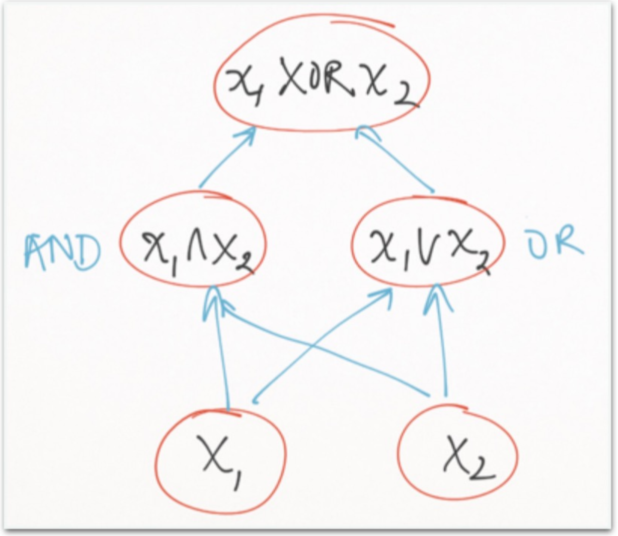


The XOR function

- Note that a perceptron cannot learn the XOR function:
- **Exercise:** come up with the parameters of a two layer network with two hidden units that computes the XOR function.
    - Here is a table with a bias feature for XOR



y	x_0	x_1	x_2
+1	+1	+1	+1
+1	+1	-1	-1
-1	+1	+1	-1
-1	+1	-1	+1



Do we gain anything beyond two layers?

EXPRESSIVE POWER OF A TWO-LAYER NETWORK

Theorem 10 (Two-Layer Networks are Universal Function Approximators). *Let F be a continuous function on a bounded subset of D-dimensional space. Then there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximate F arbitrarily well. Namely, for all $\boldsymbol{x}$ in the domain of F, $\left|F(\boldsymbol{x}) - \hat{F}(\boldsymbol{x})\right| < \epsilon$.*

- Colloquially "a two-layer network can approximate any function".
- Going from one to two layers dramatically improves the representation power of the network.

How many hidden units?

- d dimensional data with k hidden units has $(d+2)k + 1$ parameters.
  - $(d+1)k$ in the first layer (1 for the bias) and $k+1$ in the second layer
- With n training examples, set the number of hidden units $k \sim n/d$ to keep the number of parameters comparable to size of training data.
- k is both a form of regularization and inductive bias
- Training and test error vs. k

Training a two-layer network

- Optimization framework:

$$\min_{W,v} \sum_n \frac{1}{2} \left(y_n - \sum_i \mathbf{v}_i f(\mathbf{w}_i^T \mathbf{x}_n) \right)^2$$

- Loss minimization: replace squared-loss with any other.
- Regularization:
  - Traditionally NN are not regularized (early stopping instead)
  - But you can add a regularization (e.g. L_2-norm of the weights)
- Optimization by gradient descent
  - Highly non-convex problem so no guarantees about optimality.

Training a two-layer network (cont.)

- Optimization framework:

$$\min_{W,v} \sum_n \frac{1}{2} \left(y_n - \sum_i \mathbf{v}_i f(\mathbf{w}_i^T \mathbf{x}_n) \right)^2$$

- Or equivalently

$$\min_{W,v} \sum_n \frac{1}{2} \left(y_n - \mathbf{v}^T \mathbf{h}_n \right)^2$$

$$\mathbf{h}_{i,n} = f(\mathbf{w}_i^T \mathbf{x}_n)$$

- Computing gradients: second layer

$$\frac{dL_n}{d\mathbf{v}} = -\left(y_n - \mathbf{v}^T \mathbf{h}_n \right) \mathbf{h}_n$$

Training a two-layer network (cont.)

$$\min_{W,v} \sum_n \frac{1}{2}\left(y_n - \sum_i \mathbf{v}_i f(\mathbf{w}_i^T \mathbf{x}_n)\right)^2$$

- Computing gradients: first layer
  - Chain rule of derivatives

$$\frac{dL_n}{d\mathbf{w}_i} = \sum_j \frac{dL_n}{d\mathbf{h}_j} \underbrace{\frac{d\mathbf{h}_j}{d\mathbf{w}_i}}_{O \text{ if } i \neq j} \longrightarrow \frac{dL_n}{d\mathbf{w}_i} = -\left(y_n - v^T h_n\right) v_i f'(\mathbf{w}_i^T \mathbf{x_n})\mathbf{x}_n$$

also called as back-propagation

Practical issues: gradient descent

- Use online gradients (or stochastic gradients)

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{dL_n}{d\mathbf{w}}$$

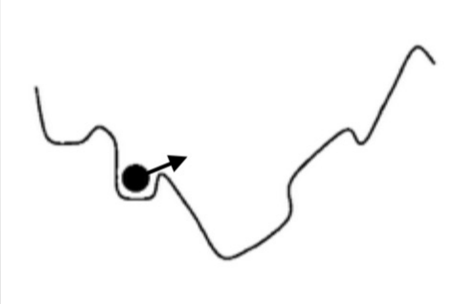
$$\underbrace{\frac{dL}{d\mathbf{w}}}_{\text{batch}} = \sum_n \underbrace{\frac{dL_n}{d\mathbf{w}}}_{\text{online}}$$

- Learning rate η: start with a high value and reduce it when the validation error stops decreasing.
- Momentum: move out small local minima

Usually set to a high value: $\beta = 0.9$

$$\Delta\mathbf{w}^{(t)} = \beta\Delta\mathbf{w}^{(t-1)} + (1-\beta)\left(-\eta\frac{dL_n}{d\mathbf{w}^{(t)}}\right)$$

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{t} + \Delta\mathbf{w}^{(t)}$$

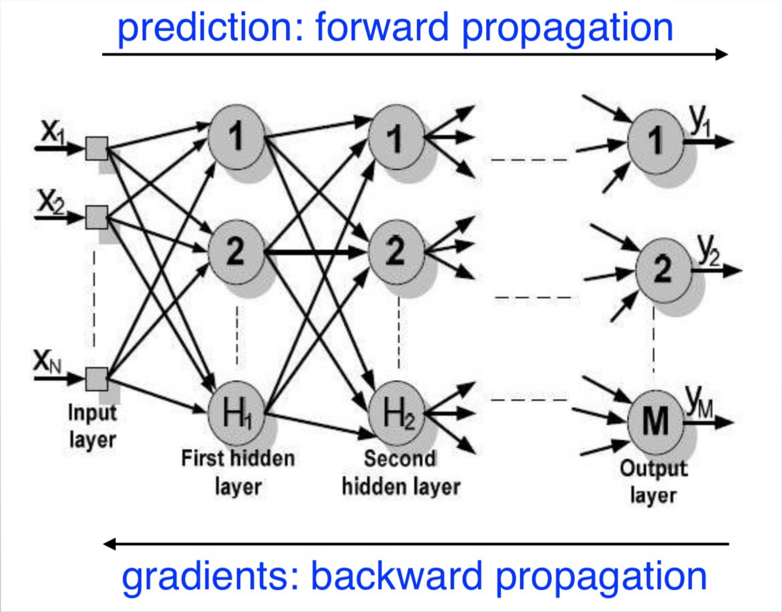


Practical issues: initialization

- Initialization didn't matter for linear models
  - Guaranteed convergence to global minima as long as step size is suitably chosen since the objective is convex
- Neural networks are sensitive to initialization
  - Many local minima
  - **Symmetries**: reorder the hidden units and change the weights accordingly to get another network that produces identical outputs
- Train multiple networks with randomly initialized weights, and pick the one that yields best validation accuracy.

Beyond two layers

- The architecture generalizes to any directed acyclic graph (DAG)
  - For example a multi-layer network
  - One can order the vertices in a DAG such that all edges go from left to right (topological sorting)
  - Gradient can be computed recursively using the chain rule.



Breadth vs. Depth

- Why train deeper networks?
- We will borrow ideas from theoretical computer science:
  - A boolean circuit is a DAG where each node is either an input, an AND gate, an OR gate, or a NOT gate. One of these is designated as an output gate.
  - Circuit complexity of a boolean function f is the size of the smallest circuit (i.e., with the fewest nodes) that can compute f.
- **The parity function**: the number of 1s is even or odd

$$\text{parity}(\mathbf{x}) = \left(\sum_d x_d\right) \mod 2$$

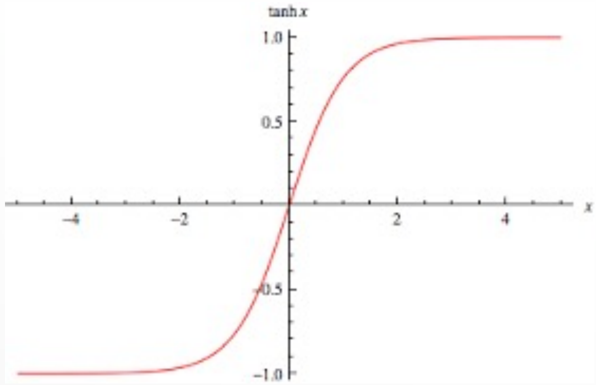
[Håstad, 1987] A depth-*k* circuit requires $\exp\left(n^{\frac{1}{k-1}}\right)$ to compute the parity function of *n* inputs

Breadth vs. Depth

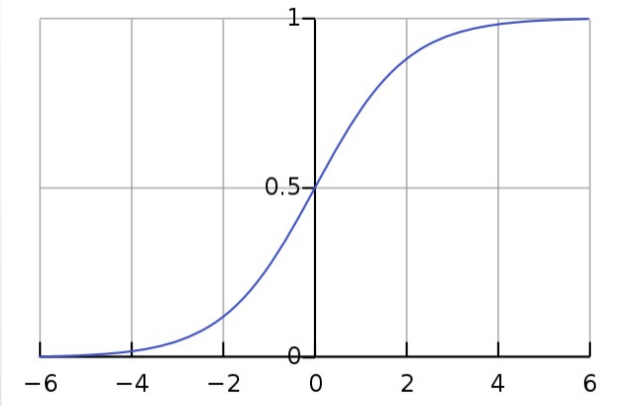
- Why **not** train deeper networks?
- Vanishing gradients
  - Gradients shrink as one moves away from the output layer
  - Convergence is slow
- But:
- Training deep networks is an active area of research.
  - Layer-wise initialization (perhaps using unsupervised data)
  - Engineering: GPUs to train on massive labelled datasets

Choices of link function

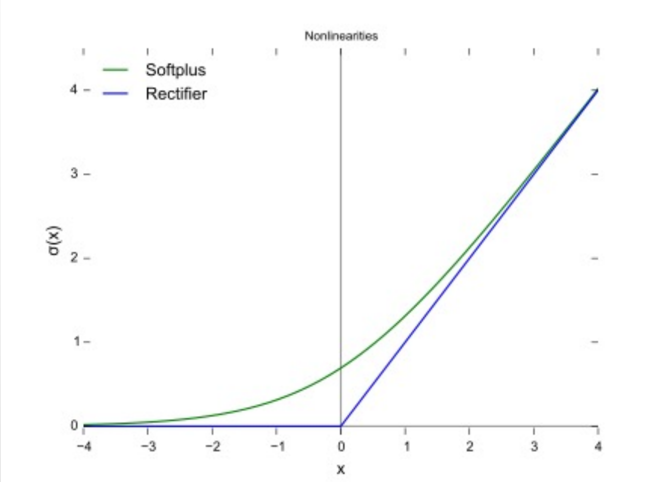
- Tanh



- Sigmoid



- Relu (rectified linear unit) (fewer vanishing gradient problems)



Other choices of the loss function

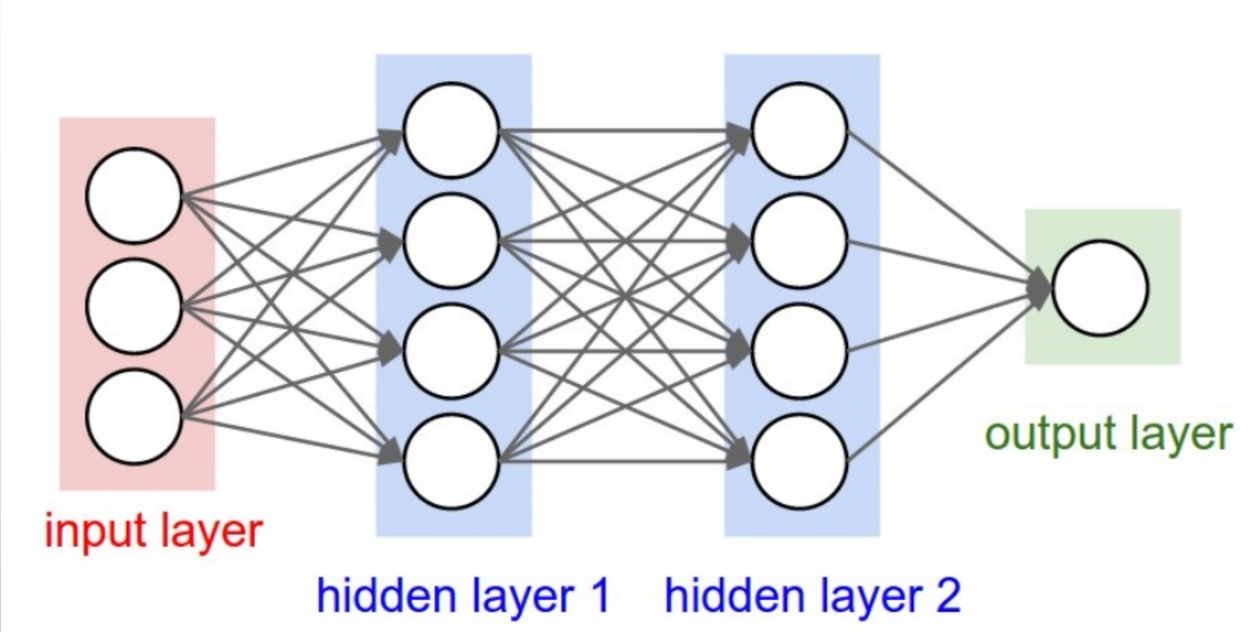
- Cross entropy (better for classification problems)
- For binary classification ($y = 0$ or 1 and f is the network output, designed to be between 0 and 1)
  - $l(y_n, f(x_n)) = -\big(y_n \log f(x_n) + (1 - y_n) \log\big(1 - f(x_n)\big)\big)$
- For multi-class problems:
  - If there are M classes, we would make M output neurons in the last layer, each designed to be between 0 and 1.
  - $l(y_n, f(x_n)) = -\sum_{c=1}^{M} \mathbb{1}(y_n = c) \log f_c(x_n)$

How to make output neurons be between 0 and 1?

- Use softmax:
- $f_j(x_n) = \frac{\exp(\beta h_j)}{\sum_{c=1}^{M} \exp(\beta h_c)}$.
- If $\beta \to \infty$, this would be equivalent with the argmax function.

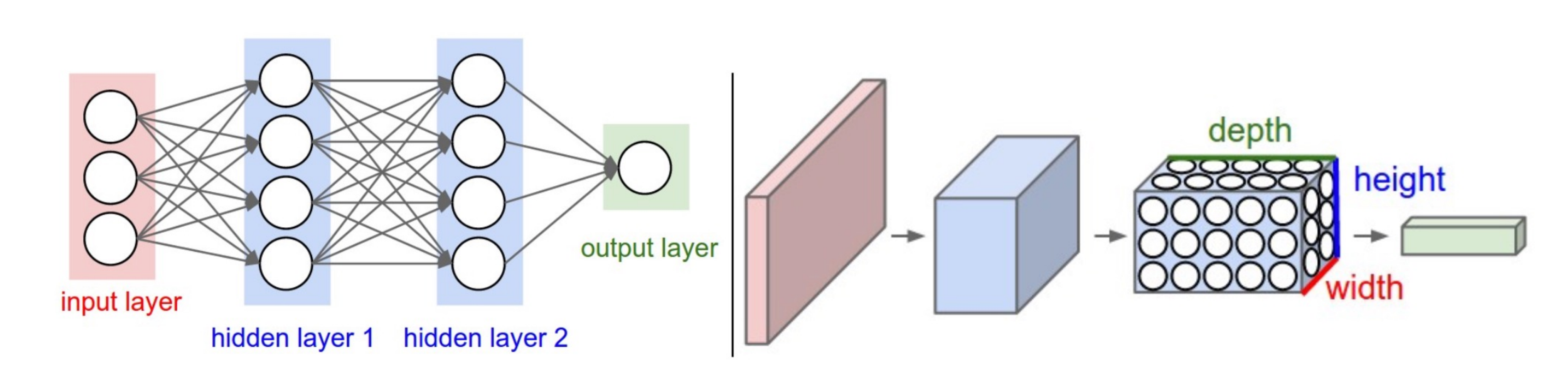
Image as input to a neural network

- *Regular neural nets don't scale well to full images.*
- Requires 32*32*3 = 3072 weights just for a single neuron in the 2nd layer.
- For larger images, number of weights increases rapidly, leading to overfitting.

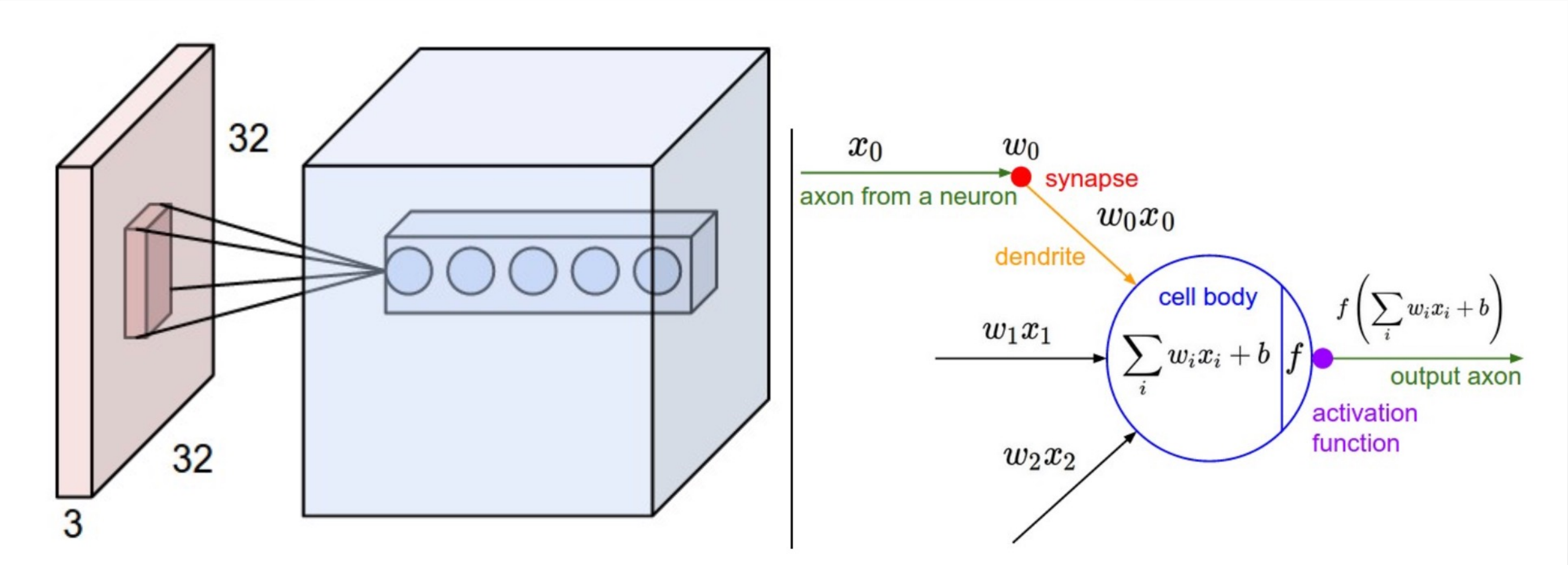


Convolutional neural nets

- The layers of a convnet have neurons arranged in 3 dimensions: **width, height, depth**.
- Each neuron in the hidden layers is only connected to a few number of neurons in the previous layer.



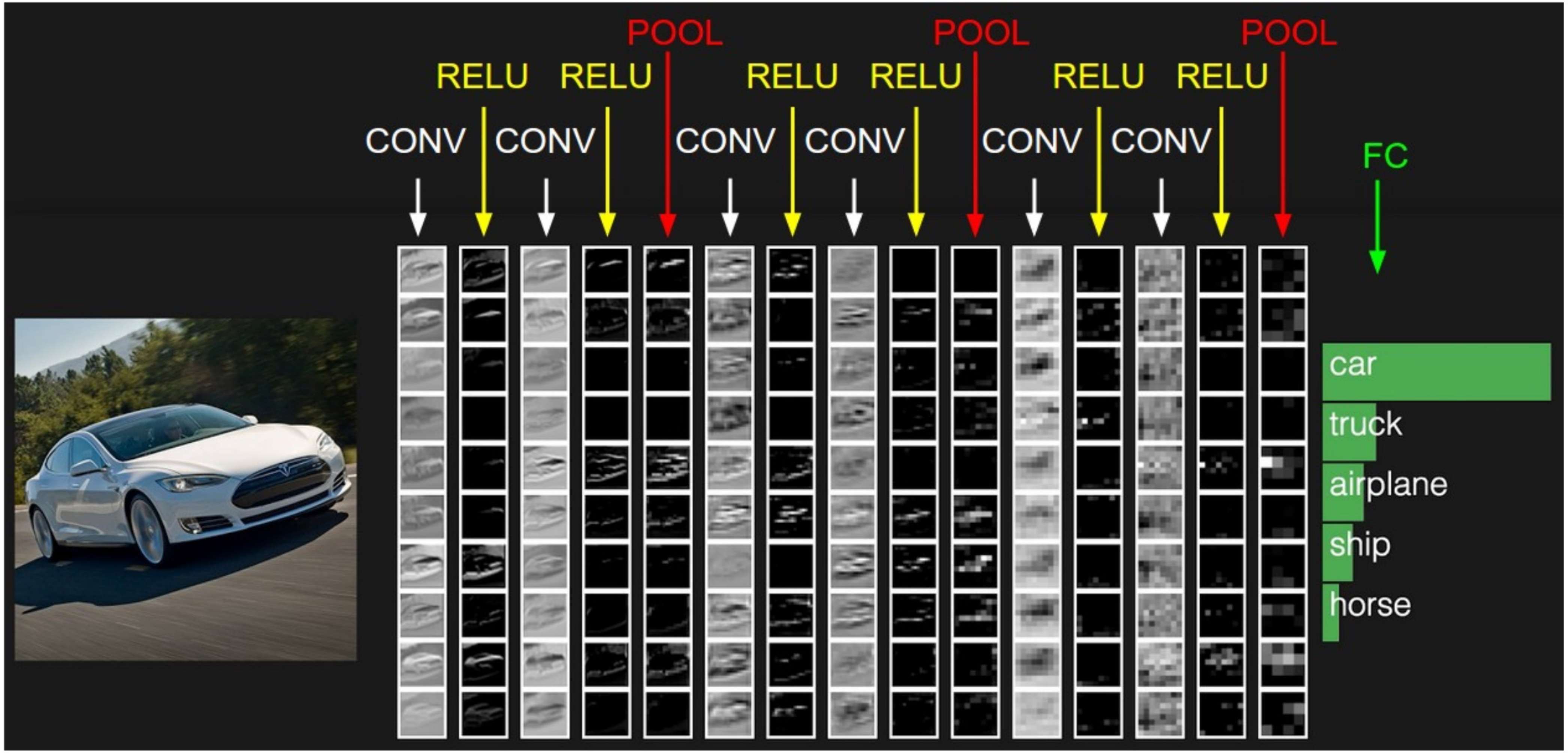
Convolutional layer



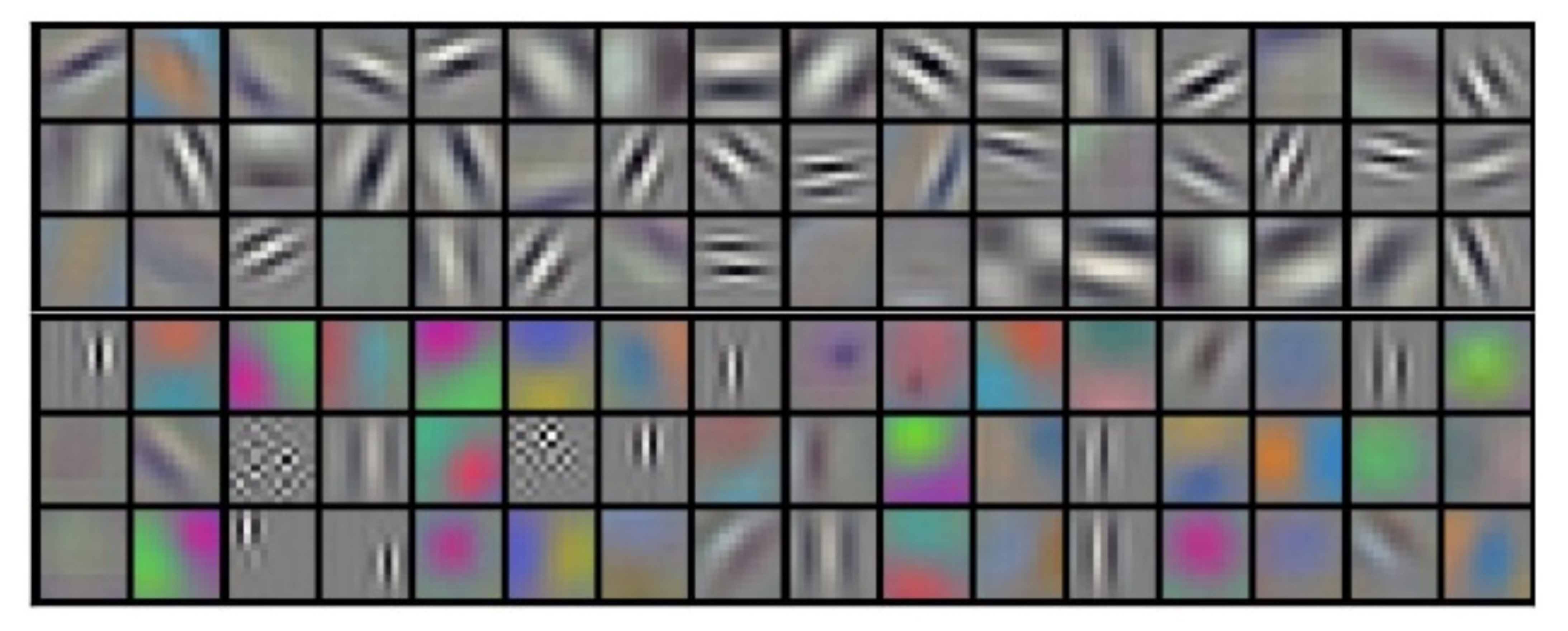
Layers used in convnets

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the $max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

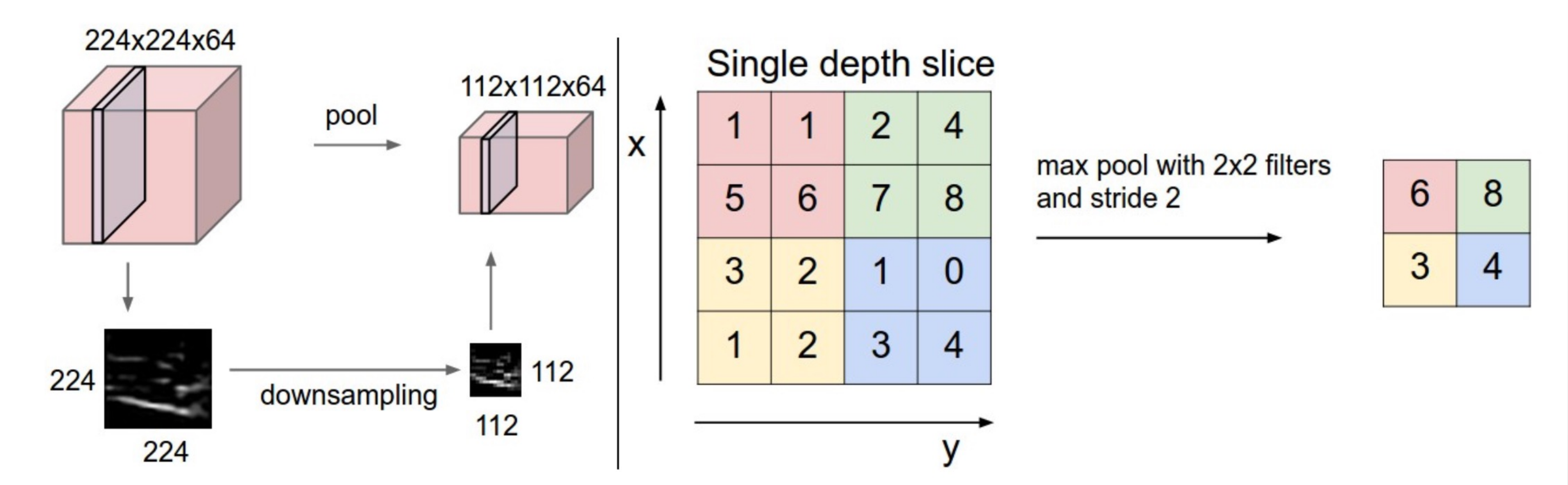
Layers used in convnets (cont.)



Examples of learned filters

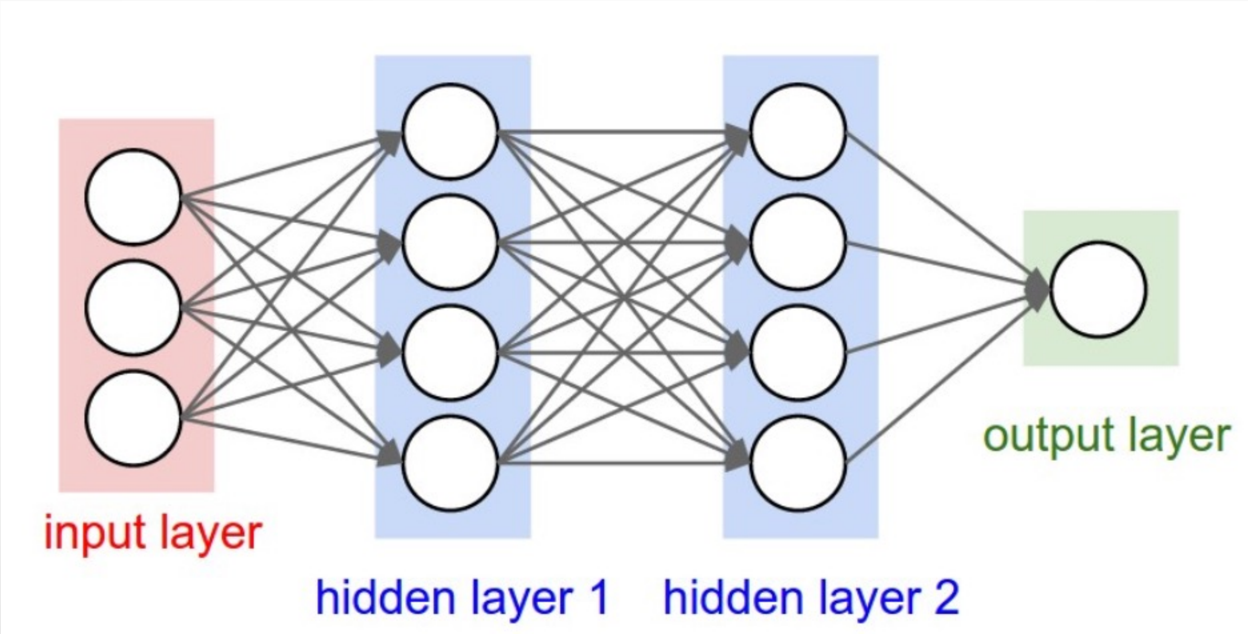


Pooling layer



Fully connected layer

- Similar to layers in multilayer perceptrons.



Layer patterns

Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transition to fully-connected layers. The last fully-connected layer holds the output, such as the class scores. In other words, the most common ConvNet architecture follows the pattern:

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```