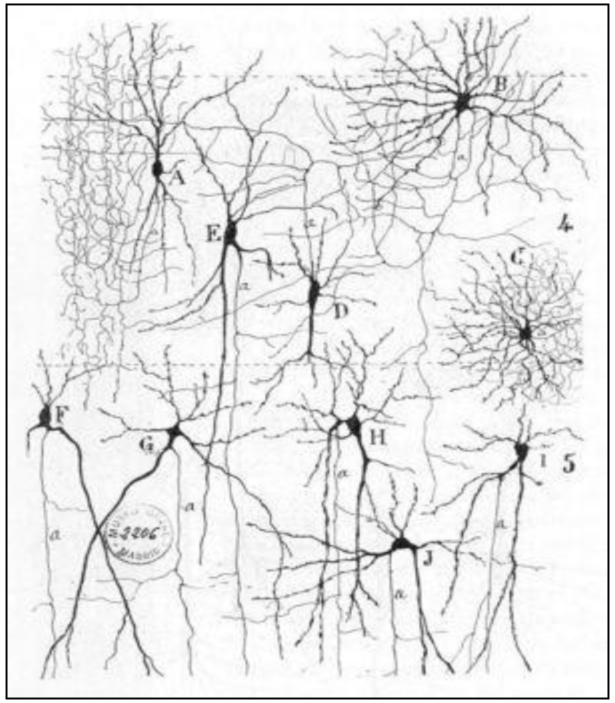
Neural Networks

What is a Neural Network?

The original idea: To make a computer "think" and "learn", maybe we should start by trying to understand how the human brain think and learn.

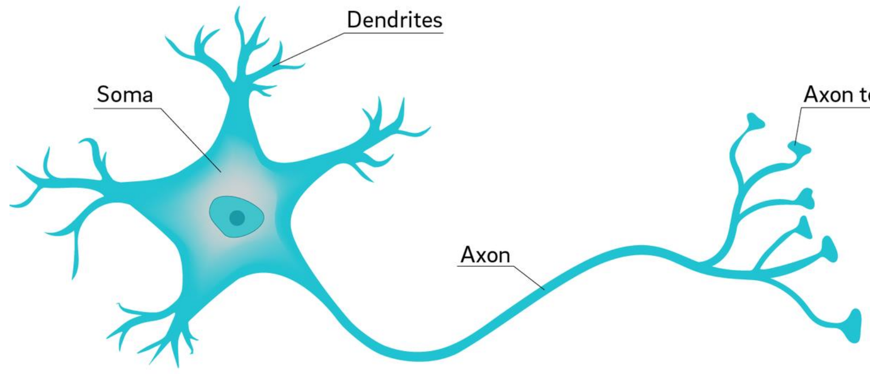


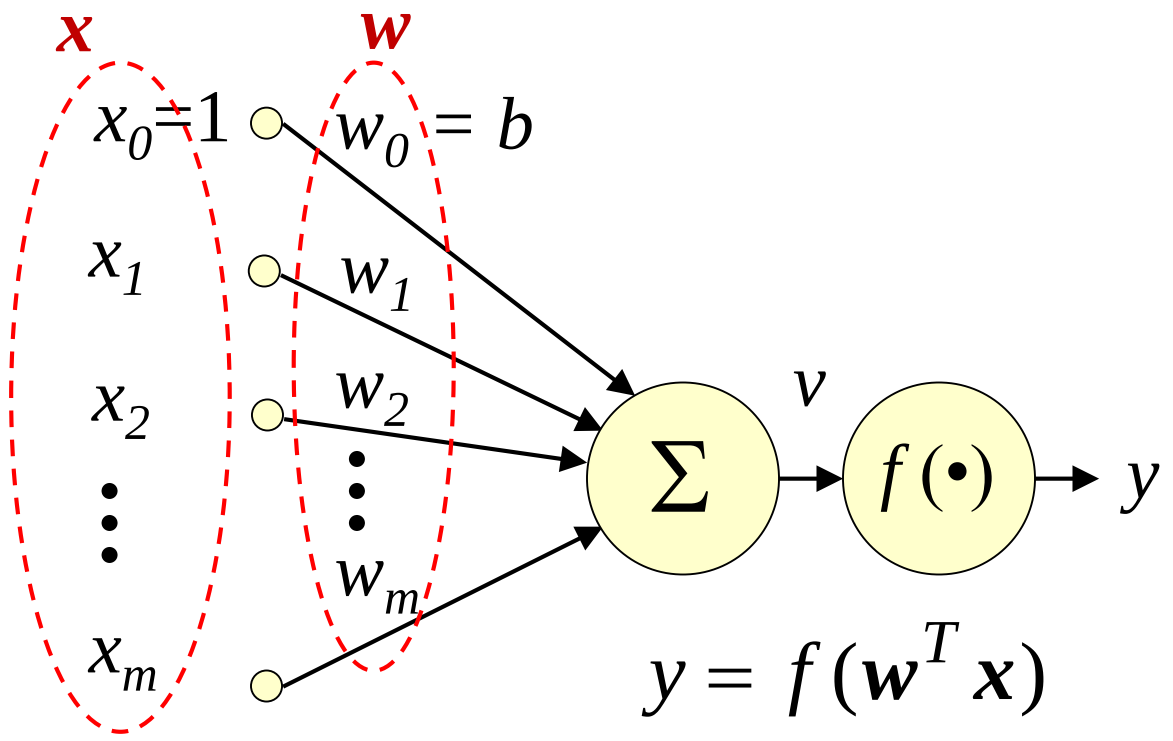
Connectionist model: Complicated functions can be described by a set of interconnected simple units.

From "Texture of the Nervous System of Man and the Vertebrates" by Santiago Ramón y Cajal. The figure illustrates the diversity of neuronal morphologies in the auditory cortex.

This is a model that spans cognitive psychology, neuroscience, philosophy, and computer science.

Remembering Artificial Neurons





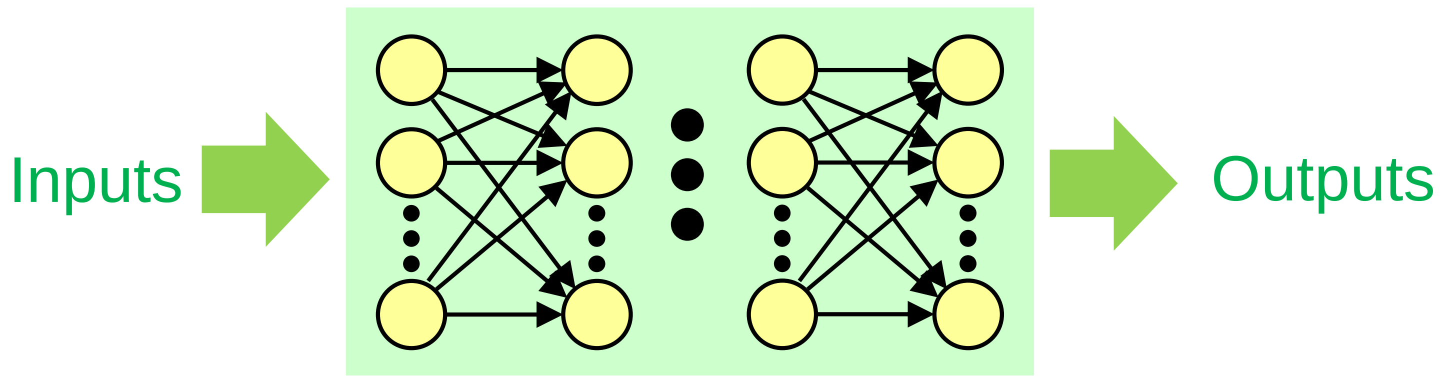
$$y = f(\boldsymbol{w}^T \boldsymbol{x})$$

Important components:

- Inputs
- Synaptic weights
- Bias
- Summing function
- Activation function
- Output

A Neural Network

An artificial neural network is made of a directed graph of a set of neurons:



A "neural network" architecture is how the neurons are connected.

Two main categories:

- Feedforward (without cycles)
- Recurrent (with cycles)

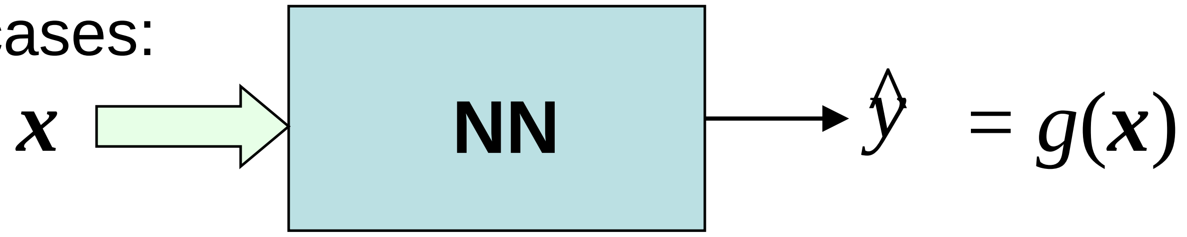
Additional variations:

- Skip connections
- Shared weights (e.g., convolutional neural networks)
- …

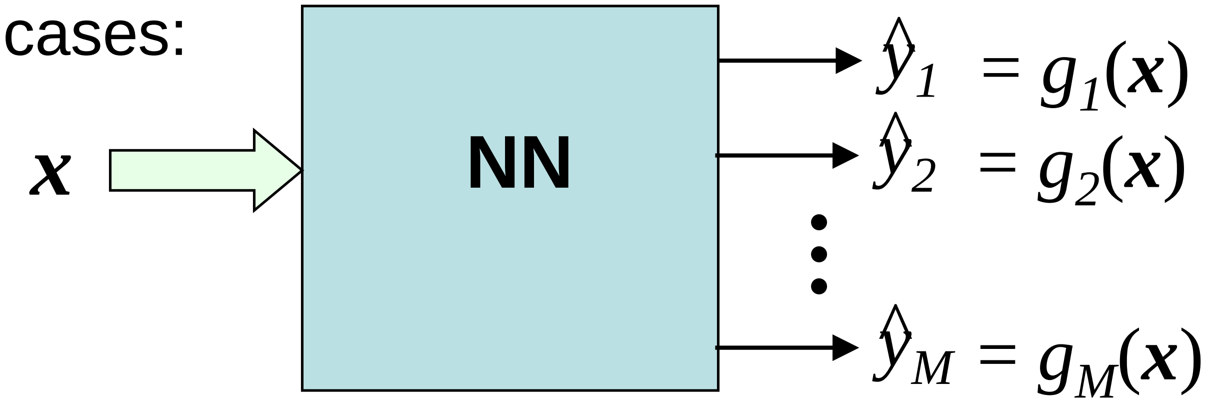
Neural Networks as Classifiers

When using neural networks as classifiers, we use their outputs as the discriminant functions.

Two-class cases:

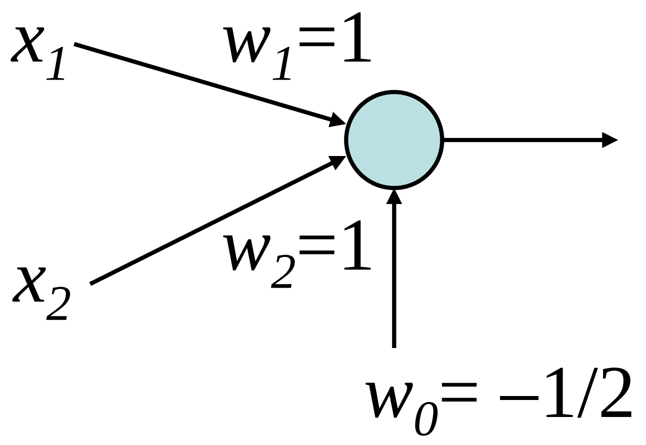


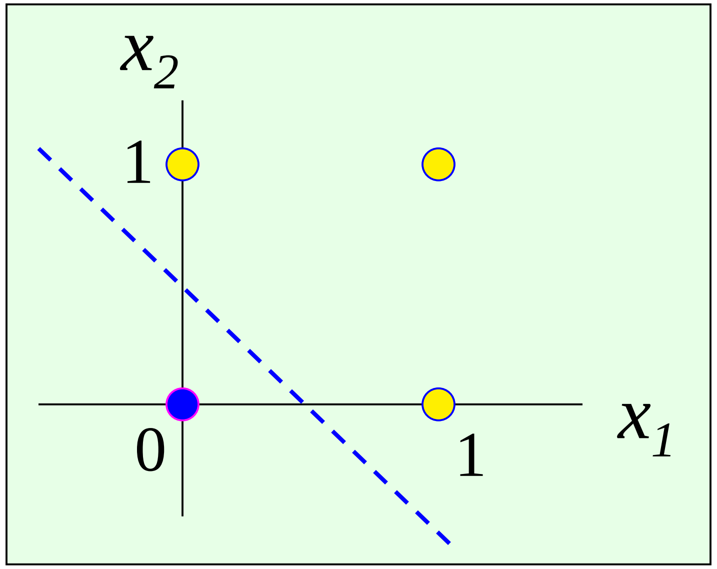
Multi-class cases:


$\boldsymbol{x}$
NN
$\hat{y}_1 = g_1(\boldsymbol{x})$
$\hat{y}_2 = g_2(\boldsymbol{x})$
$\vdots$
$\hat{y}_M = g_M(\boldsymbol{x})$

Perceptrons for Boolean Logic

Next, we use Boolean logic to illustrate the construction of neural networks as classifiers. Example: The OR operator can be implemented with a perceptron. We will just give a solution here without training:



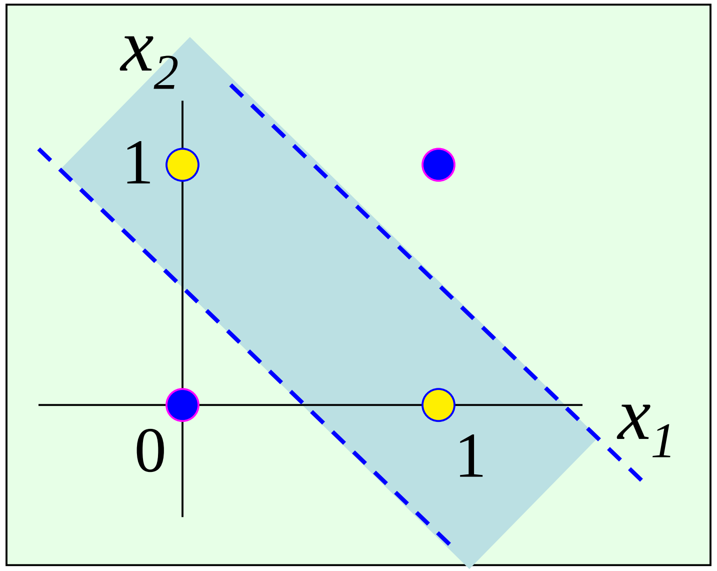


Activation function:

$f(\bullet) = 1$ (class 1; TRUE) for positive arguments
$f(\bullet) = 0$ (class 2; FALSE) for negative arguments

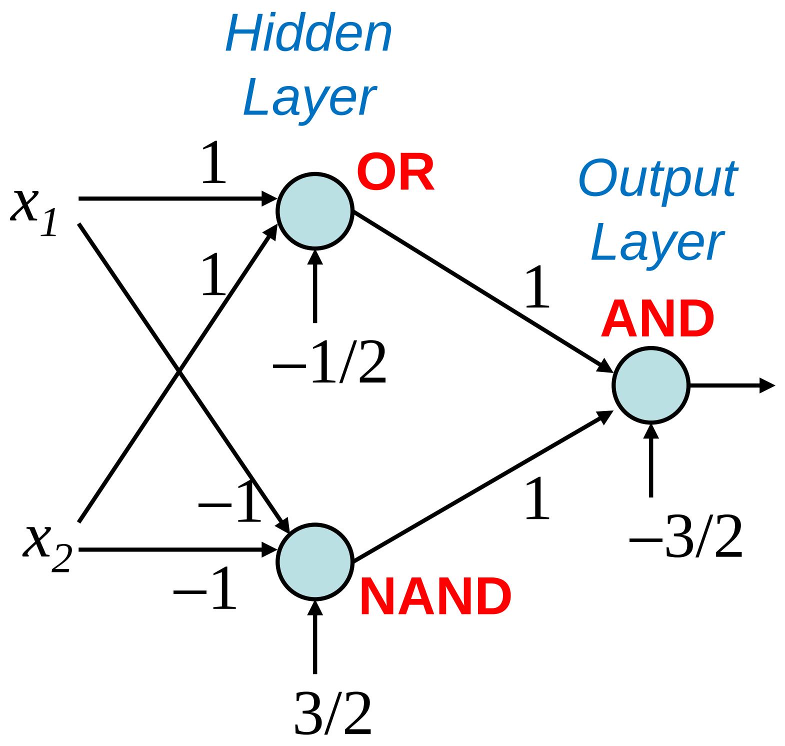
The XOR Problem

XOR is an important example of a very simple problem that is not linearly separable. However, we can get exact classification if our decision boundary includes two lines:



We can AND the regions of OR and NAND operators as the region of XOR.

This results in a two-layer perceptron:

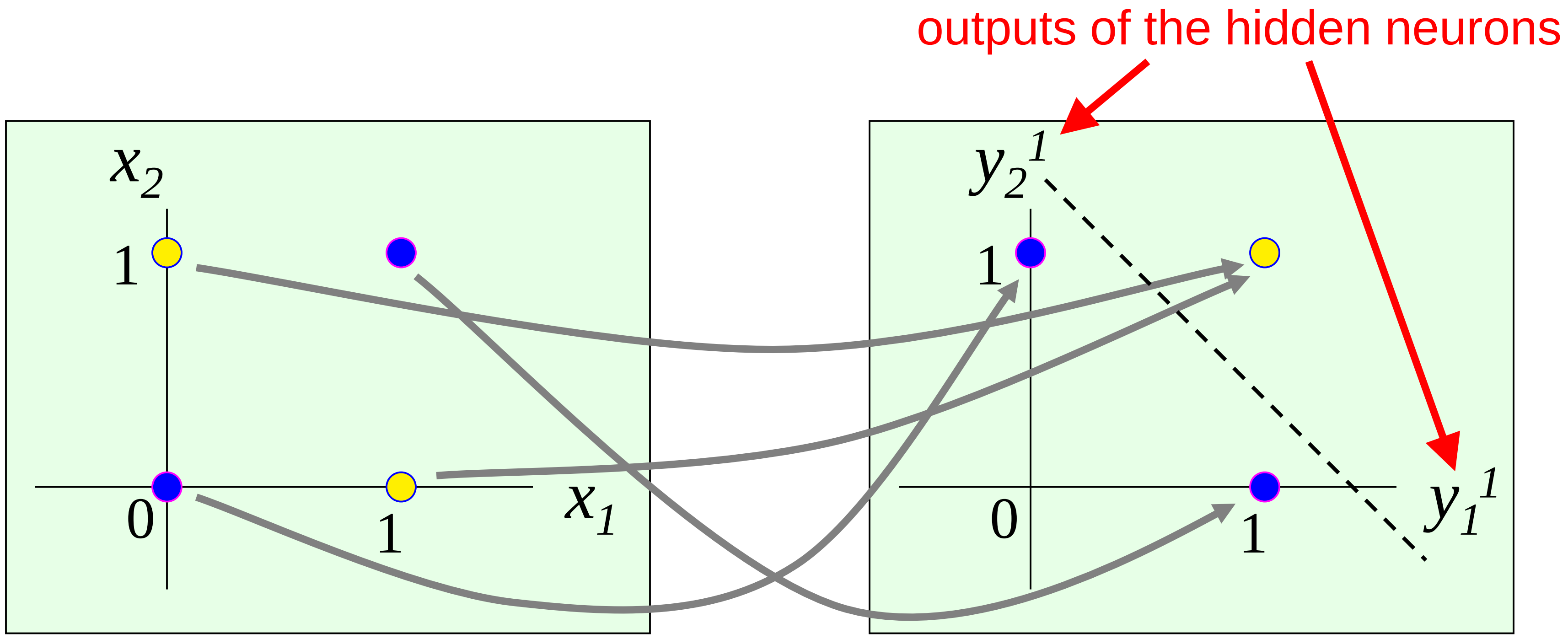


$$f(v) = \begin{cases} 1 & \text{for } v > 0 \\ 0 & \text{otherwise} \end{cases}$$

x_1	x_2	v_A	y_A	v_B	y_B	v_C	y_C
0	0	–1/2	0	3/2	1	–1/2	0
1	0	1/2	1	1/2	1	1/2	1
0	1	1/2	1	1/2	1	1/2	1
1	1	3/2	1	–1/2	0	–1/2	0

The Hidden Layer as Mapping

We can consider the hidden layer of a 2-layer perceptron as a mapping from the space of $\boldsymbol{x}$ to the space of $\boldsymbol{y}^1$ (the space of the outputs of the hidden neurons). Consider our 2-layer perceptron for the XOR problem:



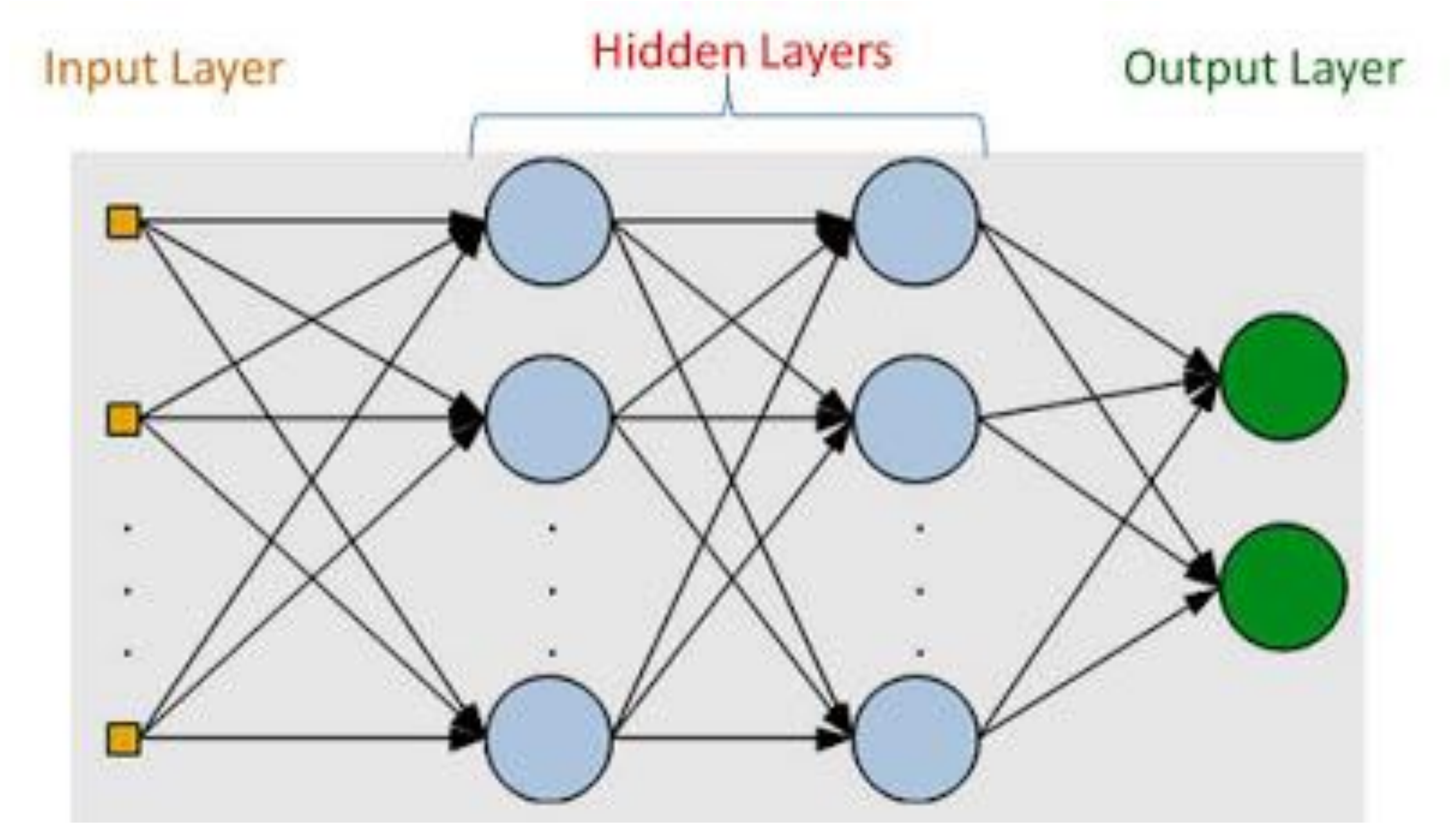
Now the two classes are linearly separable, and can be classified by the output neuron.

The Hidden Layer as Mapping

- Hidden layers of a neural network are often regarded as "feature extractors", as they tend to represent computed features that are semantically closer to the desired outputs.
- Example: In image classification tasks, lines and edges (typical features extracted by early layers in convolutional neural networks) are semantically closer to the outputs (types of objects or scenes) than the original inputs (raw pixel values).
- In a network with multiple hidden layers, they can represent different levels of abstraction of the original inputs.

Multi-Layer Perceptrons

Multi-layer perceptron (MLP) is a common name given to feed-forward fully connected neural networks.



https://kindsonthegenius.com/blog/basics-of-multilayer-perceptron-a-simple-explanation-of-multilayer-perceptron/

MLPs as Universal Approximators

An important theorem regarding the capability of MLPs:

Given

- any continuous function $g(\mathbf{x})$ defined within a compact subset S of $\mathbf{R}^l$,
- a proper nonlinear continuous activation function (such as sigmoid) for the hidden nodes,
- a linear activation function for the output node,
- enough hidden nodes,

Then

$g(\mathbf{x})$ can be approximated by the input-output function of a MLP to any precision, i.e.,

$$\text{For any given } \varepsilon > 0, \quad \exists MLP \ s.t. \left\| g(\mathbf{x}) - MLP(\mathbf{x}) \right\| < \varepsilon \quad \text{for } \mathbf{x} \in S$$

Historical Notes

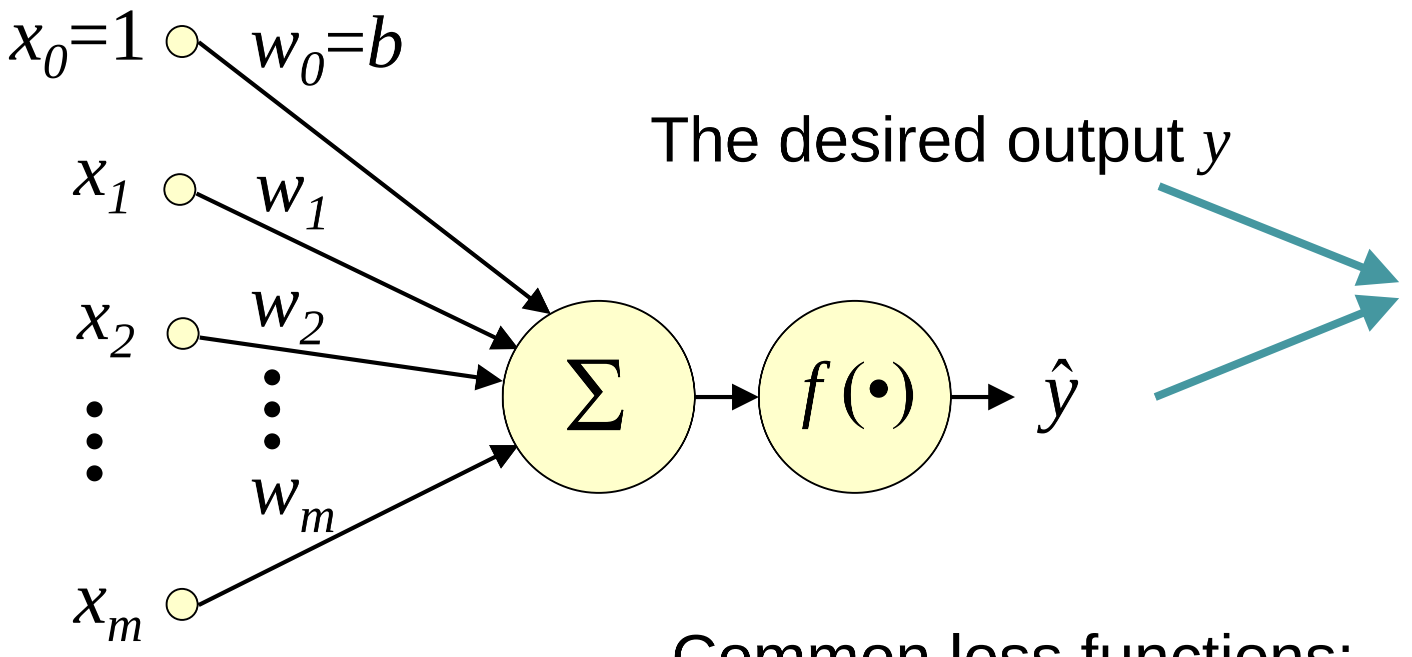
- McCulloch and Pitts (1943): A mathematic model of "neurons" based on "all-or-none" rule.
- Hebb (1949): "The Organization of Behavior": Suggesting learning through synaptic modification.
- Rosenblatt (1958): Perceptron learning, and its convergence theorem (1960). Widrow-Hoff (1960).
- Minsky (1961): Credit Assignment Problem
- Minsky and Papert (1969): Limits of single-layer perceptrons, and a wrong guess (Oops!)
- Ackley, Hinton, and Sejnowski (1985): Training MLP with simulated annealing
- Rumelhart, Hinton, and Williams (1986): Back-propagation (which had been discovered many times)

Methods for Learning in NNs

- Hebbian learning: The idea is to strengthen or weaken a connection between two neurons based on whether they are fired synchronously or asynchronously.
- Memory-based (associative) learning: The goal is to remember desired input-output pairs.
- Error-correction learning: This belongs to the category of "supervised learning". The goal is to minimize the errors (the differences between the desired and the actual outputs).. **Our focus here**
- Competitive learning: This belongs to the category of "unsupervised learning". There is no right answer. The elements (neurons in NN) "compete" to respond to the inputs.

Error-Correction Learning

This is an example with a single neuron:



The desired output y

The difference is represented by the **loss function**.

Common loss functions:

- MSE for regression problems
- Cross-entropy for classification problems

Credit Assignment Problem

There are many elements in $\boldsymbol{w}$. The problem is, how do we know how much to change each element according to the current performance?

We know how to do this in the single-layer perceptron case.

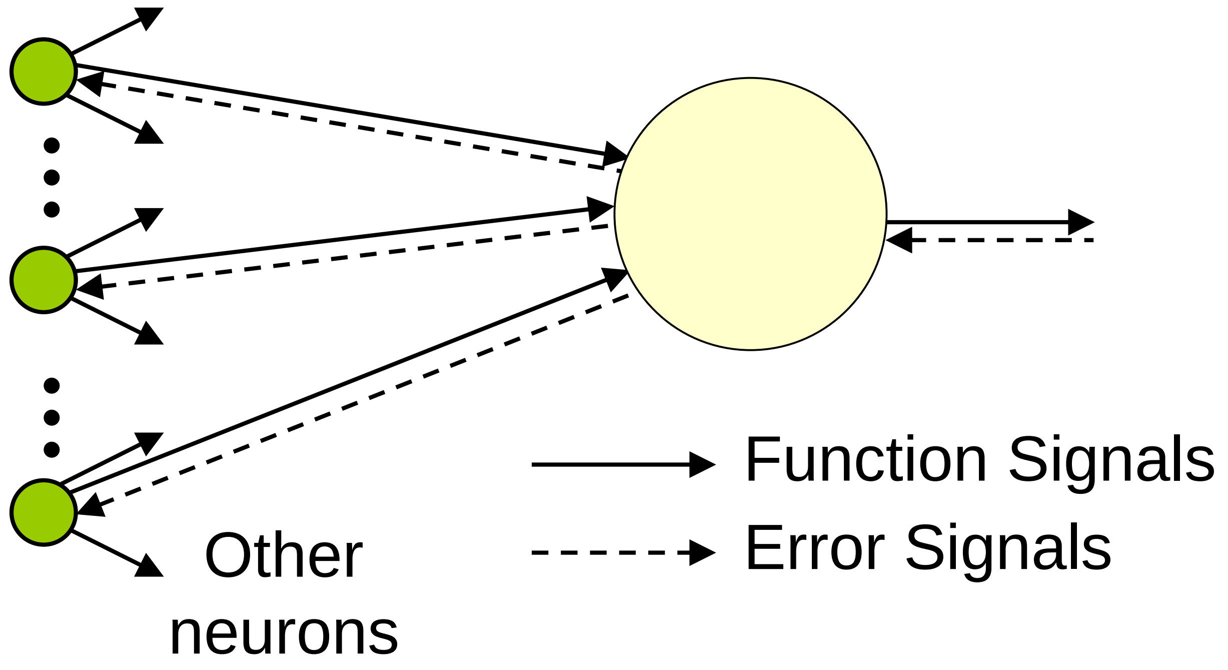
However, this is not trivial for MLP because it is too difficult to actually differentiate J relative to each component of $\boldsymbol{w}$ directly.

Signal Flows of Back-Propagation

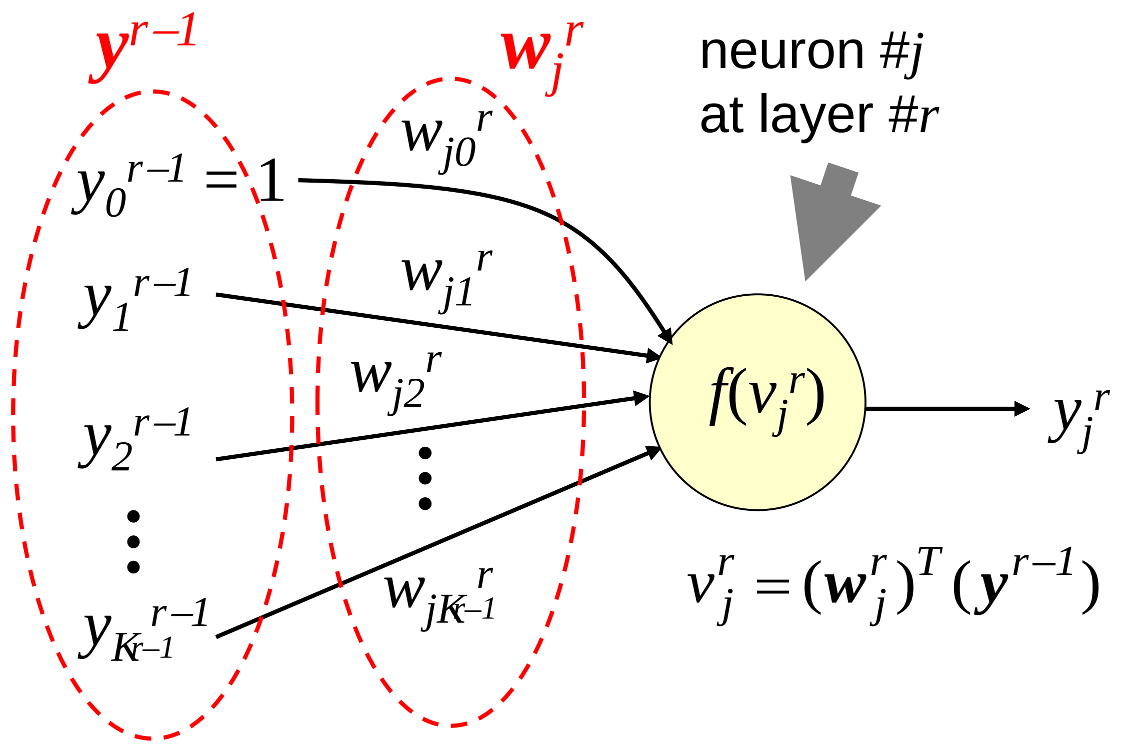
What we can do is layer-by-layer processing:

- Only update the weights from the neurons in the previous layer.
- Propagate the error to the individual neurons in the previous layer.

Signal Flow:



Notations for Multi-layer Perceptrons



$$v_j^r = (\mathbf{w}_j^r)^T (\mathbf{y}^{r-1})$$

L: # layers (hidden + output)

K_r: # neurons at layer #r ($K_0 = l$)

$\mathbf{x} = [x_1\ x_2\ \dots\ x_l]^T$: inputs (dimension l)

$\mathbf{y} = [y_1\ y_2\ \dots\ y_{K_L}]^T$: target outputs

y_k^r: output of neuron #k at layer #r

$$y_j^0 = x_j,\ 1 \le j \le l$$

$$y_0^r = 1,\ \forall r$$

$\mathbf{w}_j^r$: weight vector (dimension $(K_{r-1}+1)$) of neuron #j at layer #r

Back-Propagation

Output layer ($r = L$):

Error signals (computed from outputs)

$$\frac{\partial J}{\partial \mathbf{w}_j^L} = \left(\frac{\partial J}{\partial y_j^L}\right)\left(\frac{\partial y_j^L}{\partial v_j^L}\right)\left(\frac{\partial v_j^L}{\partial \mathbf{w}_j^L}\right) = \boxed{\left(\frac{\partial J}{\partial y_j^L}\right)} f'(v_j^L)\mathbf{y}^{L-1}$$

Error signals from the subsequent layer

Hidden layers ($r < L$):

Error signals to pass to the previous layer

$$\boxed{\frac{\partial J}{\partial y_j^r}} = \sum_{n=1}^{K_{r+1}} \left(\frac{\partial J}{\partial y_n^{r+1}}\right)\left(\frac{\partial y_n^{r+1}}{\partial v_n^{r+1}}\right)\left(\frac{\partial v_n^{r+1}}{\partial y_j^r}\right) = \sum_{n=1}^{K_{r+1}} \boxed{\left(\frac{\partial J}{\partial y_n^{r+1}}\right)} f'(v_n^{r+1})w_{nj}^{r+1}$$

$$\frac{\partial J}{\partial \mathbf{w}_j^r} = \left(\frac{\partial J}{\partial y_j^r}\right)\left(\frac{\partial y_j^r}{\partial v_j^r}\right)\left(\frac{\partial v_j^r}{\partial \mathbf{w}_j^r}\right) = \left[\sum_{n=1}^{K_{r+1}} \boxed{\left(\frac{\partial J}{\partial y_n^{r+1}}\right)} f'(v_n^{r+1})w_{nj}^{r+1}\right] f'(v_j^r)\mathbf{y}^{r-1}$$

Back-Propagation Algorithm

Update equation of weights of a neuron): $\mathbf{w}_j^r(t+1) = \mathbf{w}_j^r(t) - \rho_t \frac{\partial J}{\partial \mathbf{w}_j^r}$

This process starts from the output layer (r=L) and continues backwards through the network.

(On-line Version)

```
Initialize all the weights randomly
Repeat until the stopping criterion is met
    Choose a sample $\mathbf{x}$ from the training data
    Forward pass:
        Compute $\mathbf{y}^L$ and $J$ from $\mathbf{x}$
    Backward pass:
        Compute all $\left(\partial J / \partial \mathbf{w}_j^r\right)$
    Update all the weights
```

(Batch Version)

```
Initialize all the weights randomly
Repeat until the stopping criterion is met
    Repeat over all $\mathbf{x}$ in the training data
        Forward pass:
            Compute $\mathbf{y}^L$ and $J$ from $\mathbf{x}$
        Backward pass:
            Compute all $\left(\partial J(i) / \partial \mathbf{w}_j^r\right)$
        Accumulate (over $\mathbf{x}$) the total $\left(\partial J(i) / \partial \mathbf{w}_j^r\right)$
    Update all the weights
```

Online vs. Batch Versions

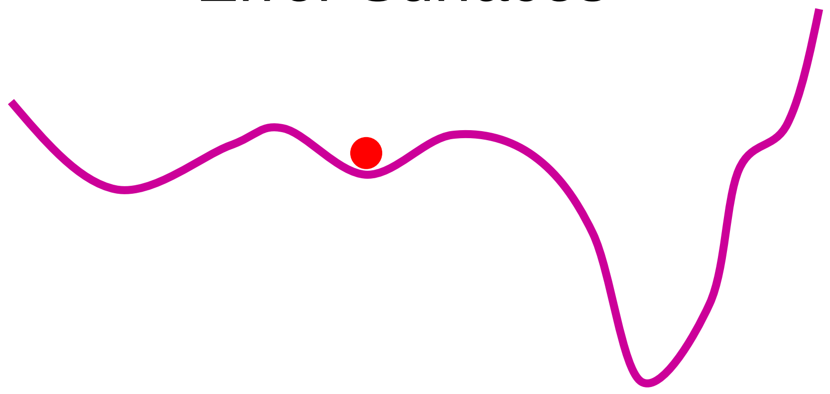
- The online version is usually called **stochastic gradient descent** today.
- The batch version has the same "error surface" ($J(\mathbf{w})$) throughout the training process.
- For the online version, the error surface is different for every training sample. This is why it is considered stochastic. (The ordering of samples should be randomized for each epoch.)
- The online version, being stochastic, is less likely to get trapped in shallow local minimums of J.
- The batch version leads to more stable learning behavior and smoother convergence. But the online version usually converges faster (or with less computation).
  - The use of mini-batches is to get the benefits of both approaches.

Online vs. Batch Versions

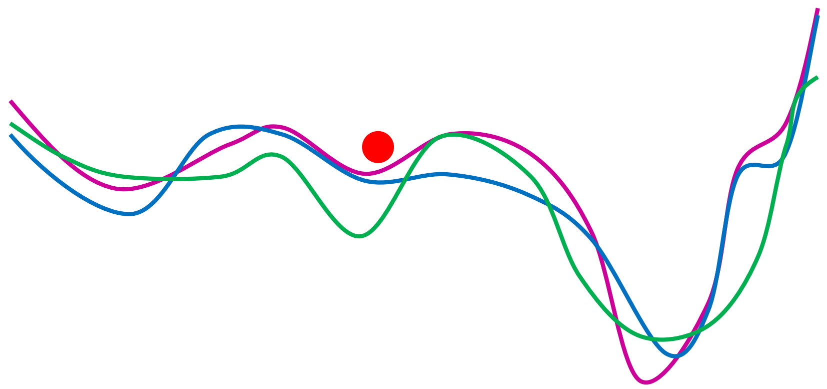
An illustration of why online training is less likely to get trapped in local minimums of the error function.

Error Surfaces

Batch Training



Online Training

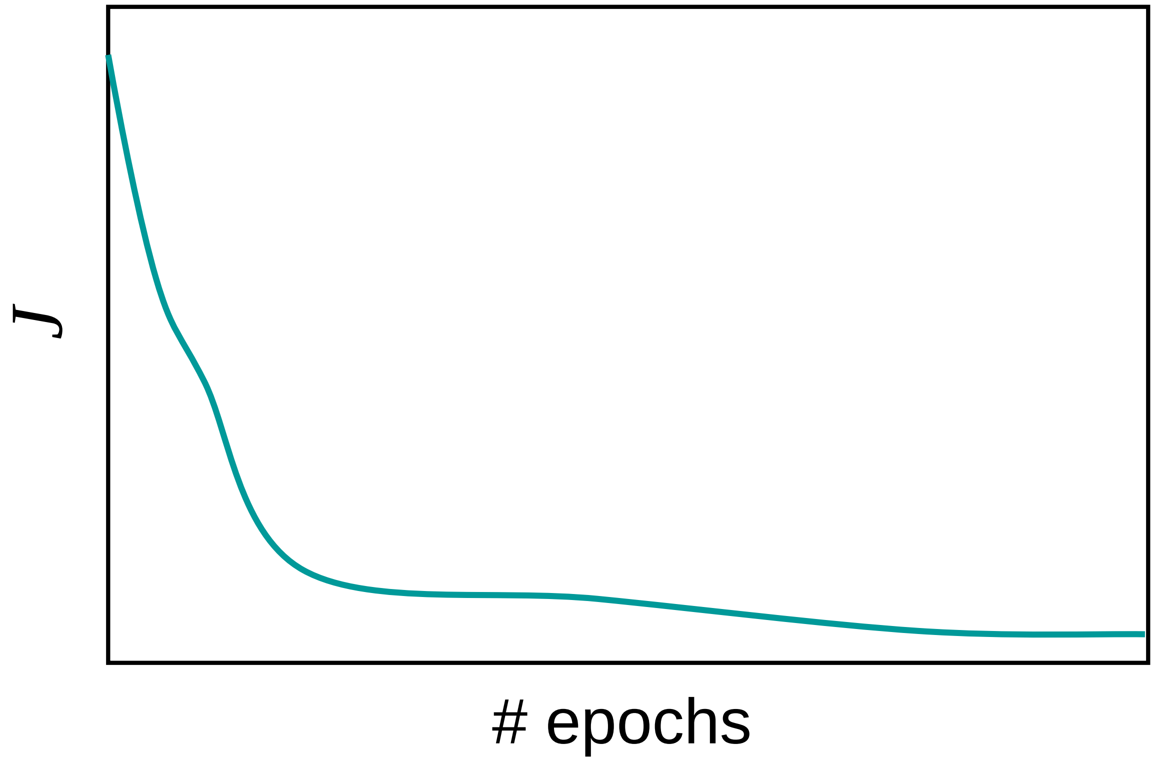


Each curve for a different sample

Learning Curve

Learning curve: The curve of J versus the number of epochs while training a NN. (An ***epoch*** is a training iteration in which all the training samples are presented once.)

A typical learning curve



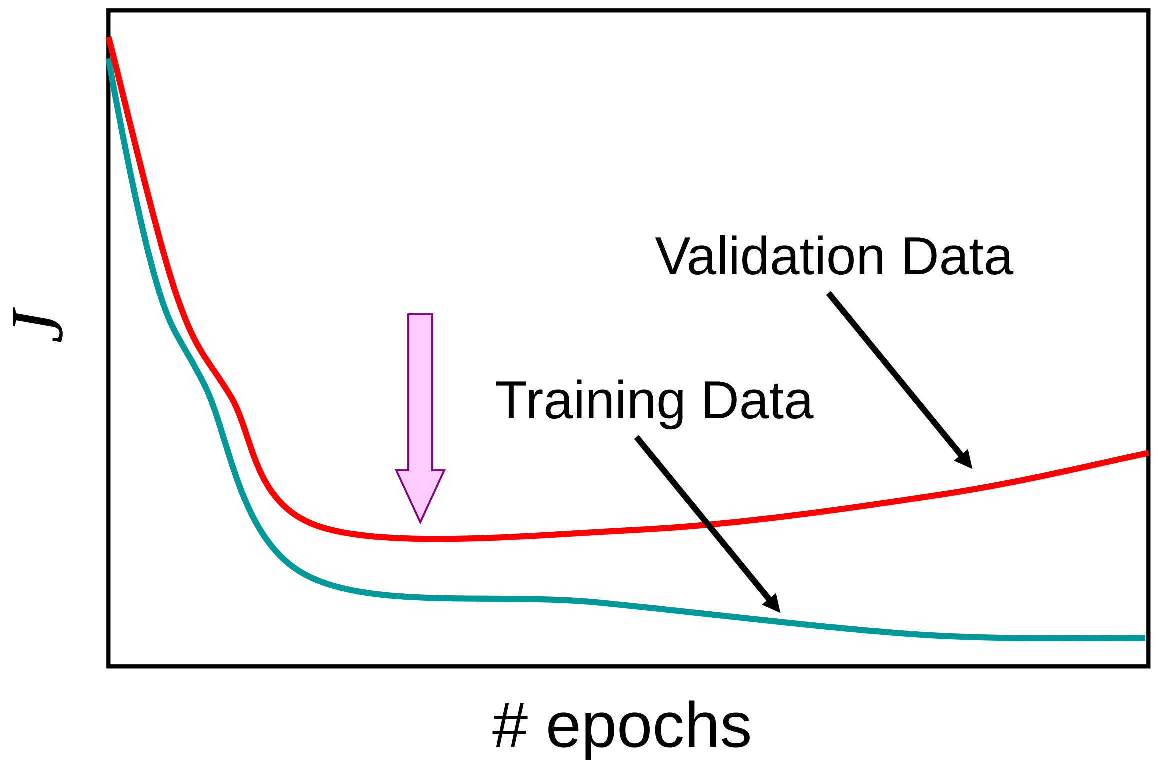
Issues in Training

There are many issues in designing and training a NN:

- Nonlinearity (through activation functions)
- Speed of convergence:
  - Normalization (data normalization, batch normalization)
  - Learning rate adjustment
  - Momentum
- Generalization (performance for data not used for training)
  - Stochastic training / dropout
  - Regularization
  - Early stopping

Early Stopping (for NN Training)

When training neural networks, we can stop the training at the minimum of J for the validation set.



Typical learning curves for the training and validation sets.

Regularization

- Include the **bias-variance** trade-off during the learning process.
- **Regularization**: Modify the learning objective to minimize BOTH the bias and the model complexity. (Normally, the learning process only attempts to minimize bias.)
- Examples:

regularization term

  - L2 regularization: $Loss = E + \lambda \sum w^2$

    ➔ weight decay in gradient descent

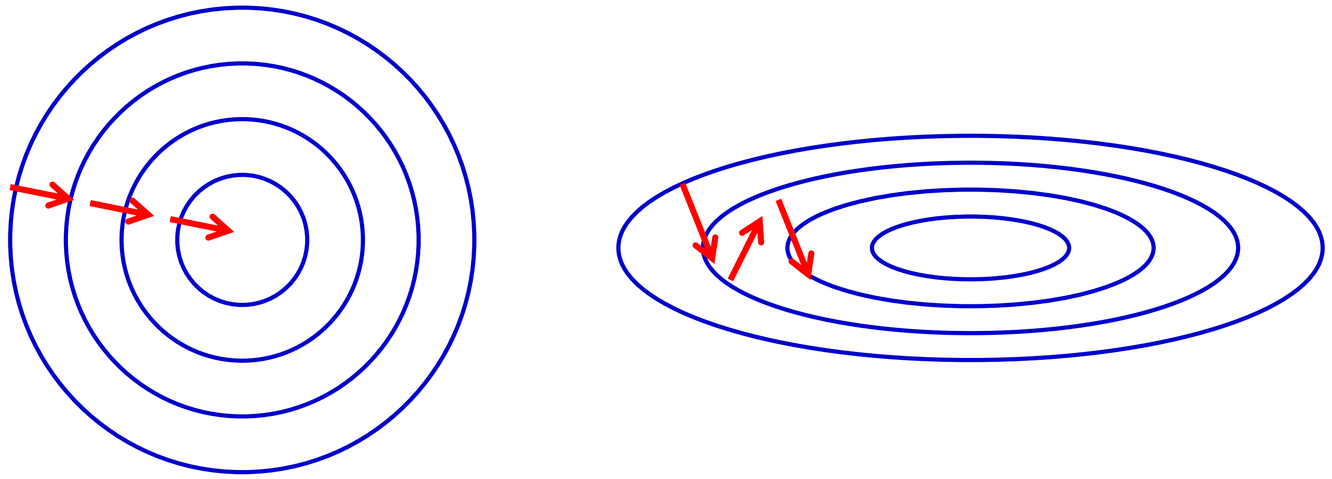
  - L1 regularization: $Loss = E + \lambda \sum |w|$

    ➔ more sparse network weights

Data Normalization

Objective: Make the different input features have similar distributions.

A simple visualization of how data normalization affects convergence speed:



How a weight vector might move during training:

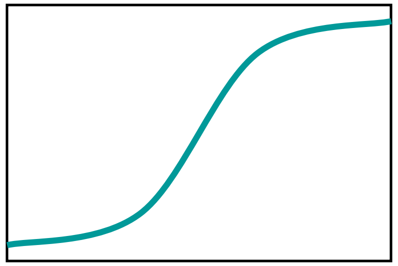
Batch Normalization

- The original data normalization only applies to the inputs of the first layer.
- Inputs to the other hidden layers can also benefit from normalization. How do we do that?
  - Those "inputs", which are outputs from previous layers, keep changing during the training.
  - Idea: Use results of "mini-batches" from the previous layer to compute the normalization transform.
- Add learnable linear transform coefficients to each BN layer.
- At inference time, the normalization transform can be computed from all the training samples.

Nonlinear Activation Functions

For a neuron, the activation function is the only source of nonlinearity. Typical activations are:

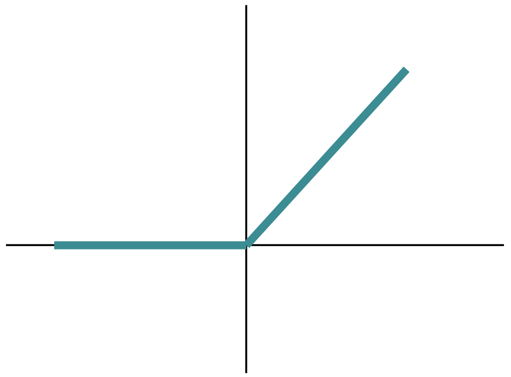
- Sigmoid/logistic and tanh:
  - The old standard
  - Continuously differential everywhere
  - Problem: Vanishing gradients



$$f(v) = \frac{1}{1+\exp(-av)}$$

$$f(v) = \tanh(av)$$

- RELU (rectified linear units):
  - The new standard for deep NNs
  - Faster gradient computation
  - Some variants
  - Nonlinear with no vanishing gradients



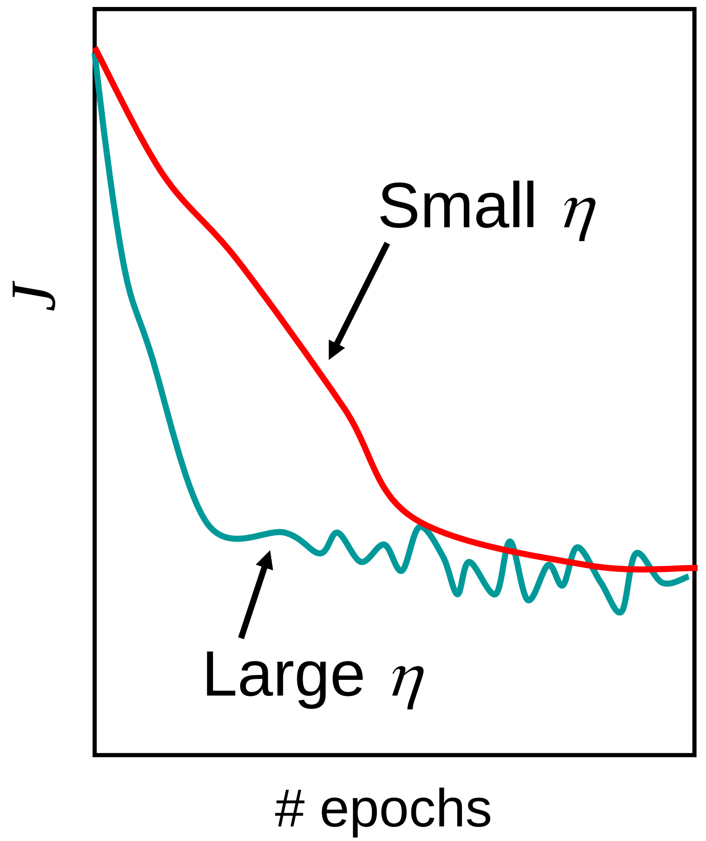
Learning Rate Adjustment

Large learning rate:

- Faster training (reaches the local minimum of J in fewer epochs)
- More oscillation

Small learning rate:

- Slower training
- Less oscillation (smoother approach to the local minimum of J)



An approach is to start with a larger learning rate (e.g., 0.1; smaller for DNNs) and gradually decrease it.

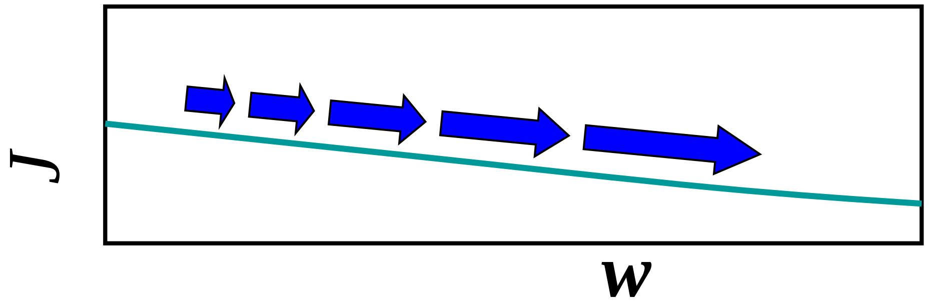
Momentum

Momentum is a popular technique for speeding up training and avoiding oscillation.

$$0 \leq \alpha < 1$$

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \frac{\partial J}{\partial \mathbf{w}} + \underbrace{\alpha[\mathbf{w}(t) - \mathbf{w}(t-1)]}_{\text{momentum term}}$$

The effect of momentum in a plateau region:



Effects of momentum:

- Acceleration: Increased update when the gradients are in similar directions.
- Stabilization: Reduced update when the gradients have opposite directions.

Adaptive Learning Rate and Momentum

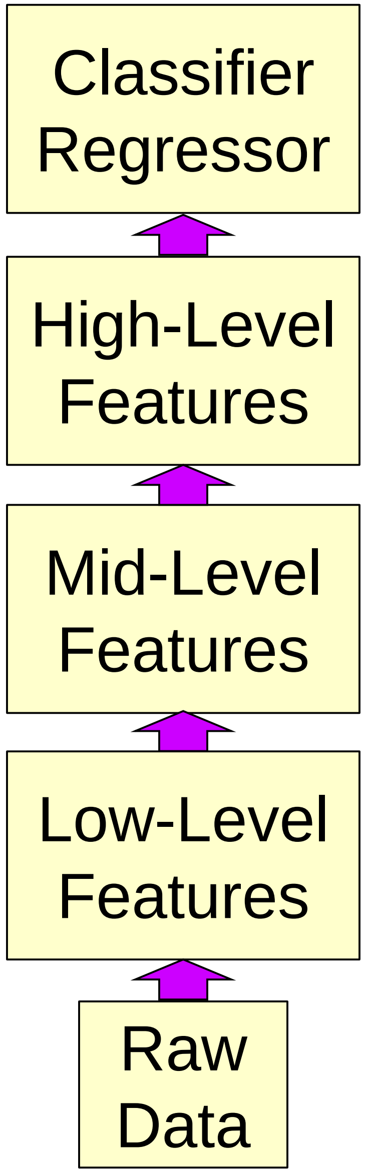
- AdaGrad: Adjust the effective learning rate for each learnable parameter separately. Parameters that have larger (smaller) cumulative previous gradient magnitudes get reduced (increased) effective learning rates.
- RMSprop: Similar to AdaGrad, but the "cumulative previous gradient magnitudes" are replaced by their running averages.
- Adam: Combining concepts from RMSprop and momentum.
  - Update term includes past updates (momentum).
  - Update term scaled by running average of past gradient magnitudes (RMSprop).

Convolutional Neural Networks

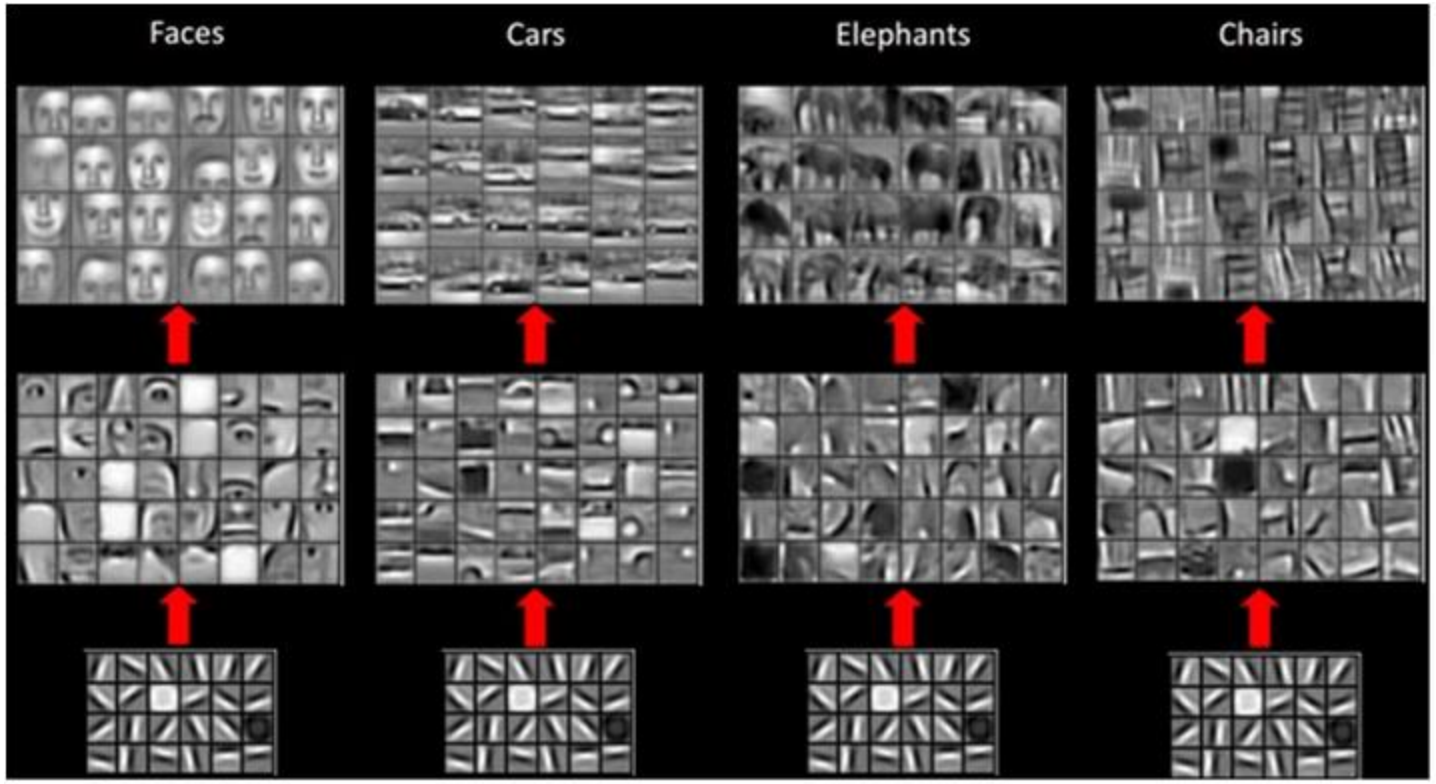
Let's assume the use of image data here:

- The "convolution" operation: Compute a feature for a pixel by applying a "weighted sum" of the pixel values in its neighborhood.
- The weights represent a "mask", "kernel", or "filter".
- The same kernel is applied to all the pixels. (Example: Gradient is one of the most common kernel.)
  - This is sometimes called weight sharing, and it helps to limit the overall network complexity even if we have numerous nodes (to represent the results of all the pixels.)
- There can be many kernels used ➔ many features.
- The kernel weights can be trained. (They are the weights of the CNN.) the learning process optimizes the kernel weights for the given task.
- The convolutional layers can be stacked to form feature maps of different levels.

Convolutional Neural Networks



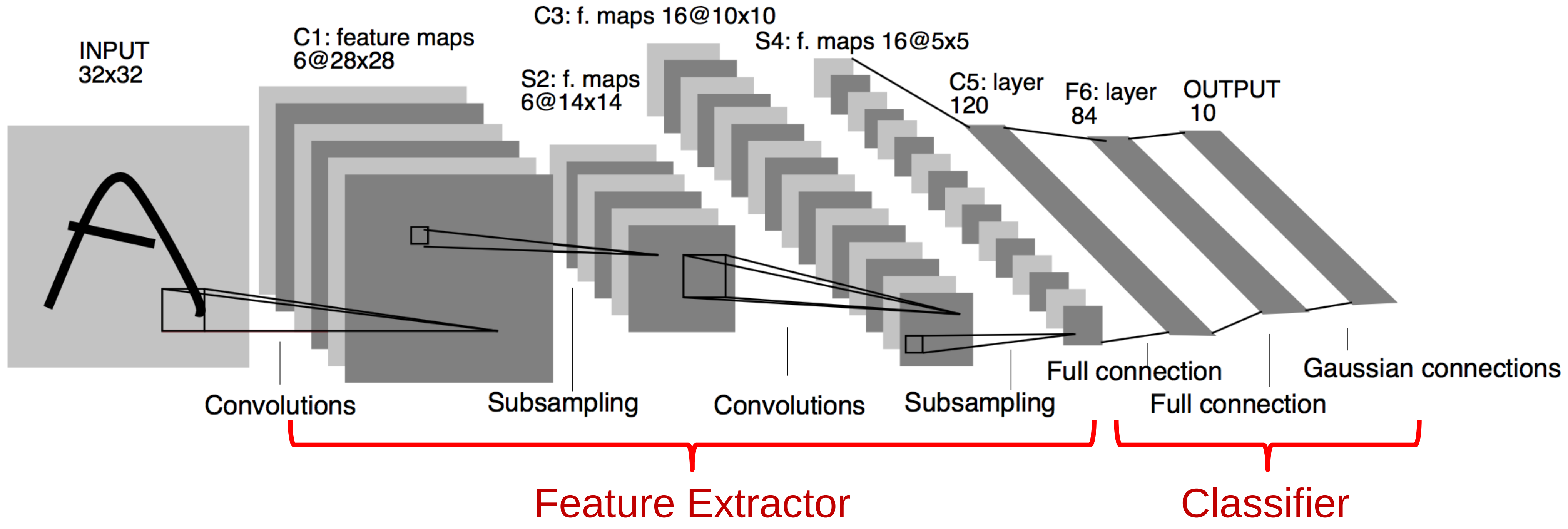
This is how our brain processes visual inputs.



https://www.researchgate.net/figure/Pictorial-representation-of-features-in-3-layers-of-a-CNN-34-Notice-the-increasing_fig2_308883811

Convolutional Neural Networks

LeNet: A small CNN example for image classification:



- The last part (fully connected layer) is actually the classifier. It is just like a traditional multi-layer perceptron.
- The CNN can also work with other classifiers (e.g., SVMs), but using FC layers is common as the training can be done end-to-end.