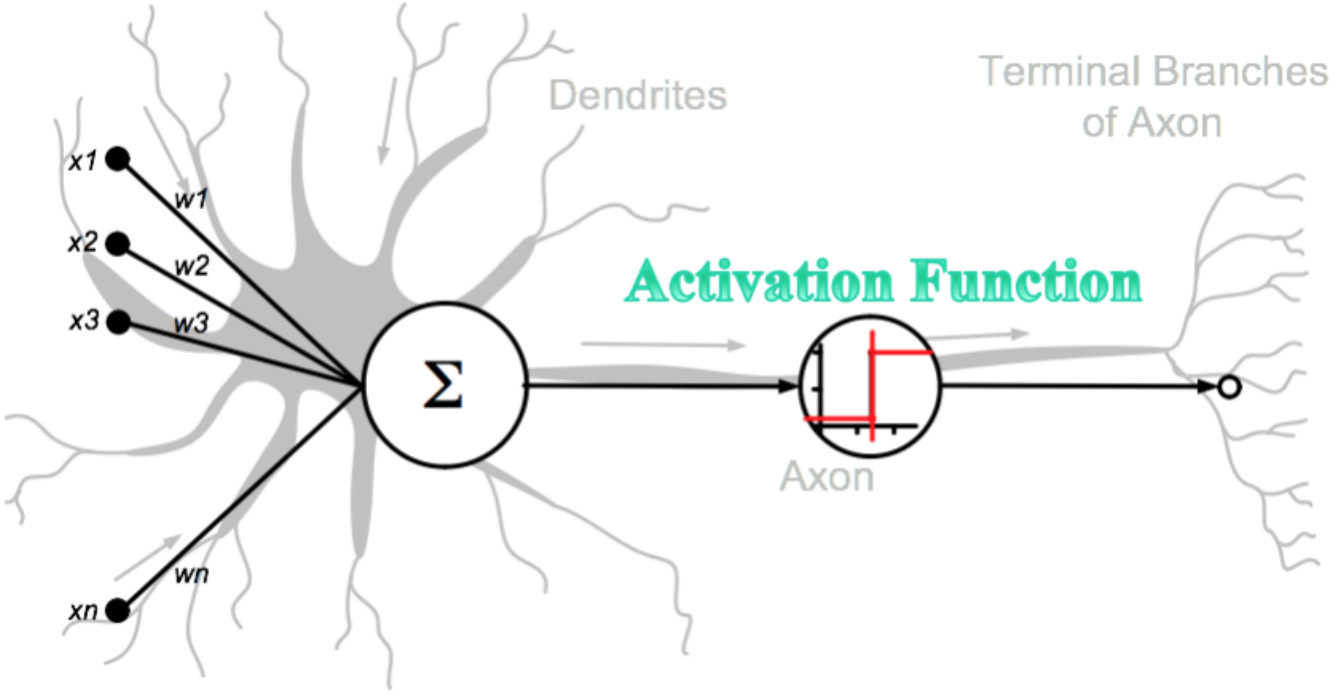
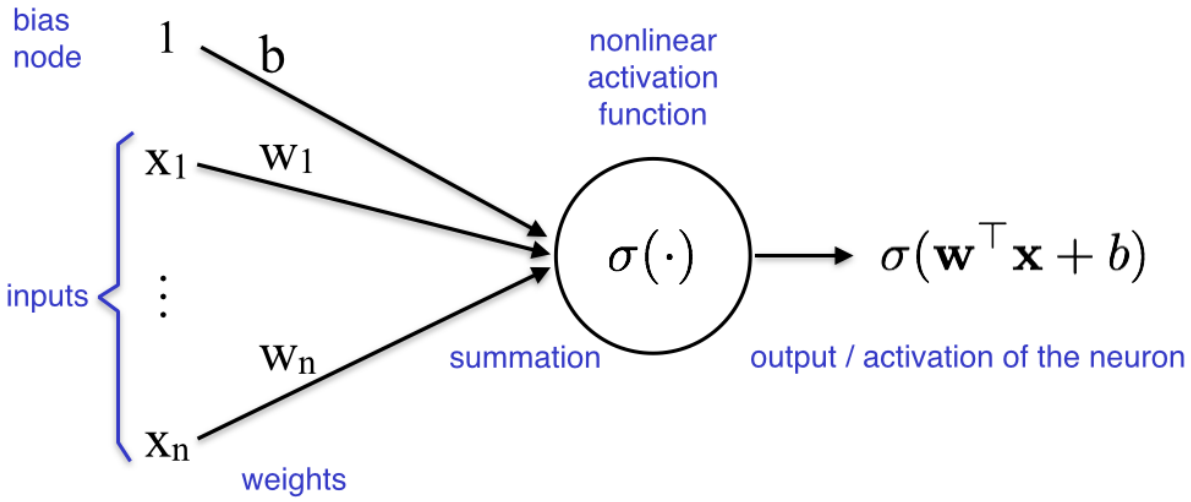
Perceptrons and Neural Networks

F. Noé[1]

Deep Learning Classes, FU Berlin 2018


Dendrites
Terminal Branches
of Axon
x1
x2
x3
xn
w1
w2
w3
wn
Σ
Activation Function
Axon



- Input vector $\mathbf{x} \in \mathbb{R}^n$
- Trainable weights: $\mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$
- Fixed nonlinear activation function $f : \mathbb{R} \to \mathbb{R}$
- Output / activation: $y = \sigma(\mathbf{w}^\top \mathbf{x} + b)$

Perceptron

F. Rosenblatt: The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Phys. Rev. 65, 386-408 (1958)

Single neuron that acts as a binary classifier with

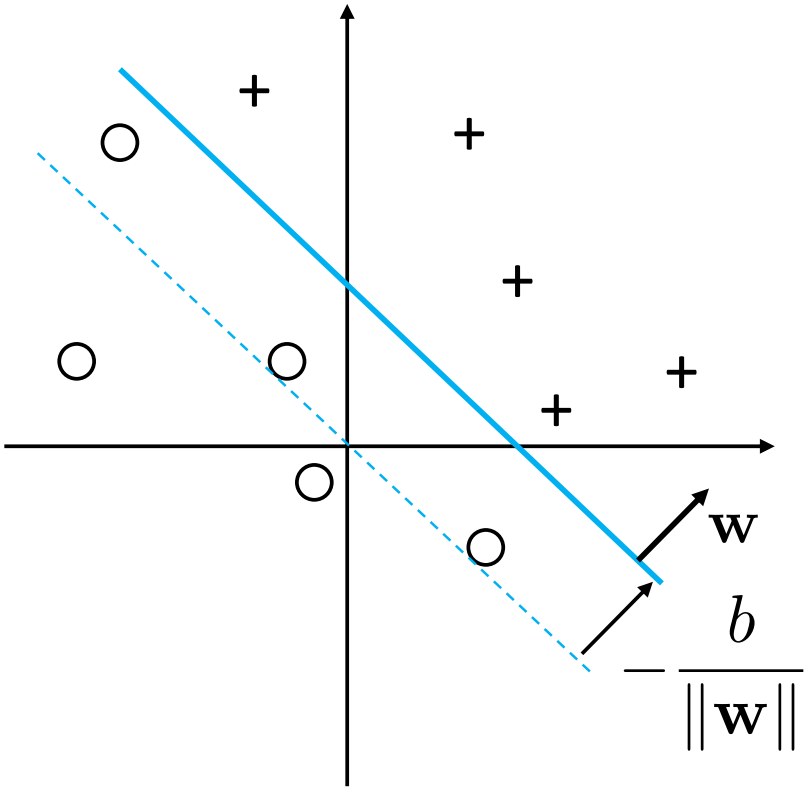
$$\sigma(x) = \begin{cases} 1 & \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{otherwise.} \end{cases}$$

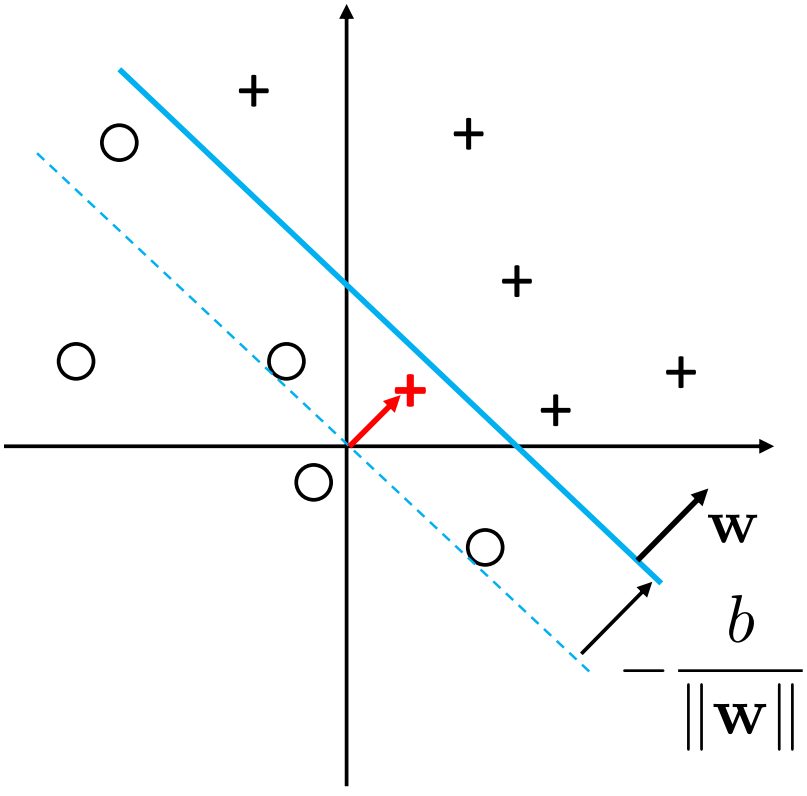
Training data $\mathbf{X} \in \mathbb{R}^{N \times n}$, $\mathbf{y} \in \{0,1\}^N$. Parameters $\mathbf{w} \in \mathbb{R}^n$, $b \in \mathbb{R}$.

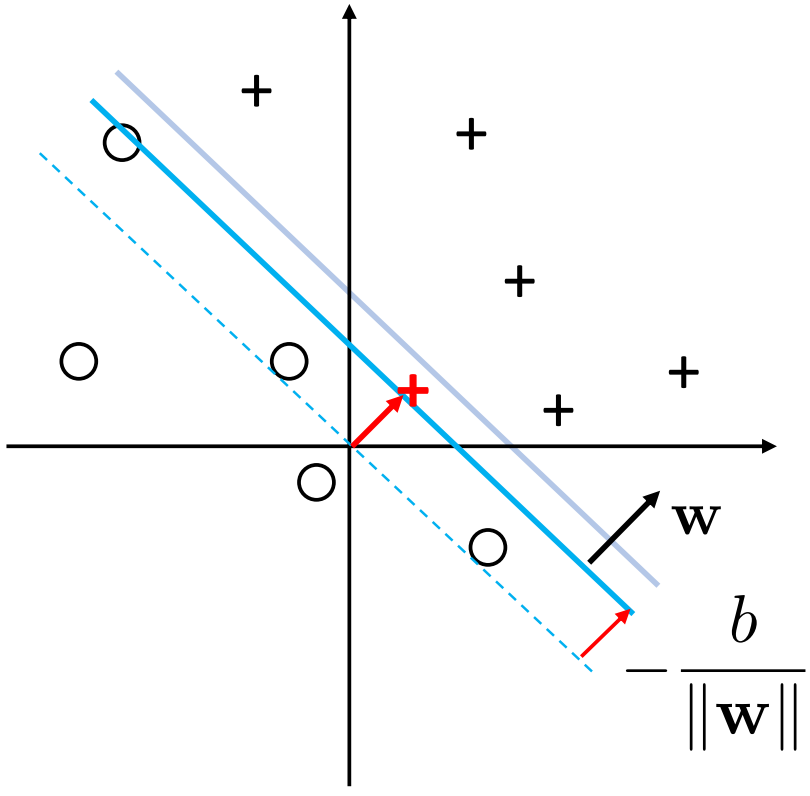
Algorithm

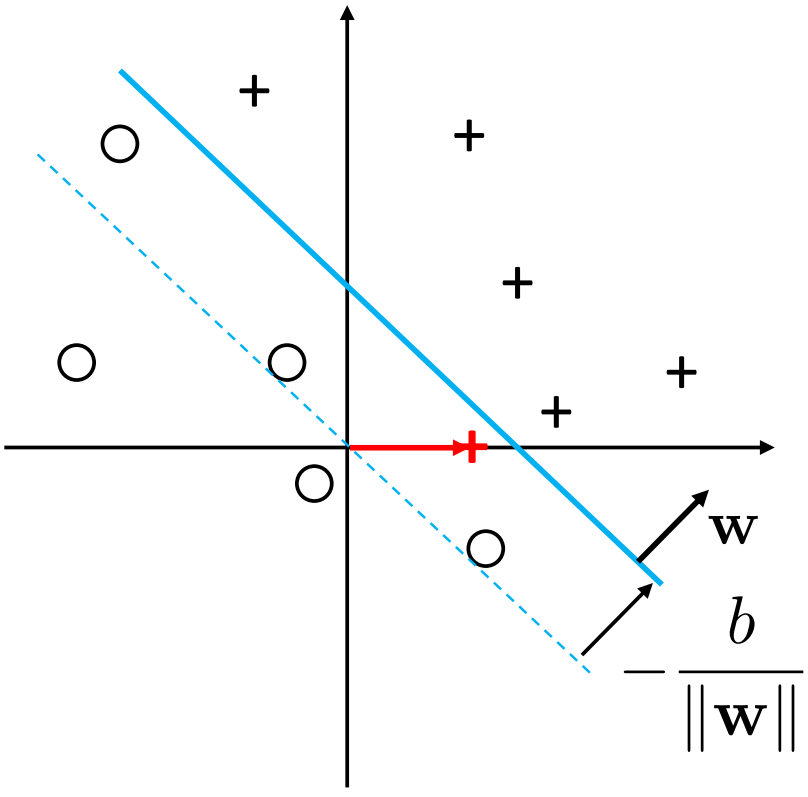
1. Initialize $\mathbf{w}^0$ and b^0
2. For each training pair $(\mathbf{x}_i, y_i)$:
   1. Predict output: $\hat{y}_i^t = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
   2. Update weights:

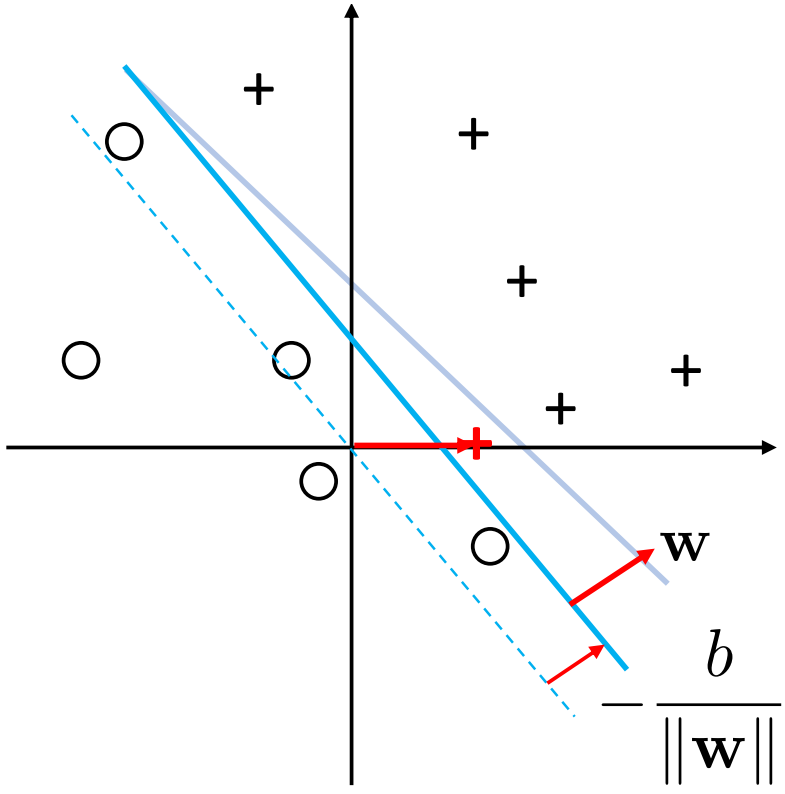
$$\mathbf{w}^{t+1} = \mathbf{w}^t + (y_i - \hat{y}_i^t)\mathbf{x}_i$$
$$b^{t+1} = b^t + (y_i - \hat{y}_i^t)$$


w
$-\frac{b}{\|\mathbf{w}\|}$

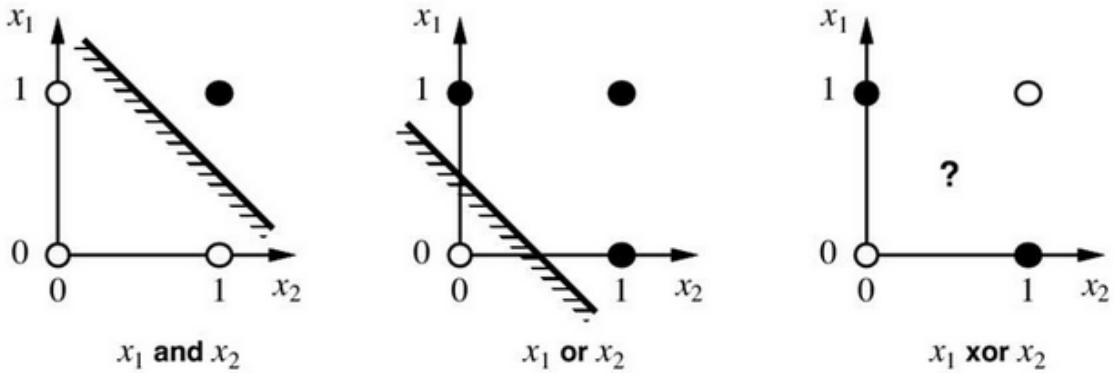

w
$-\frac{b}{\|\mathbf{w}\|}$


w
$-\frac{b}{\|\mathbf{w}\|}$

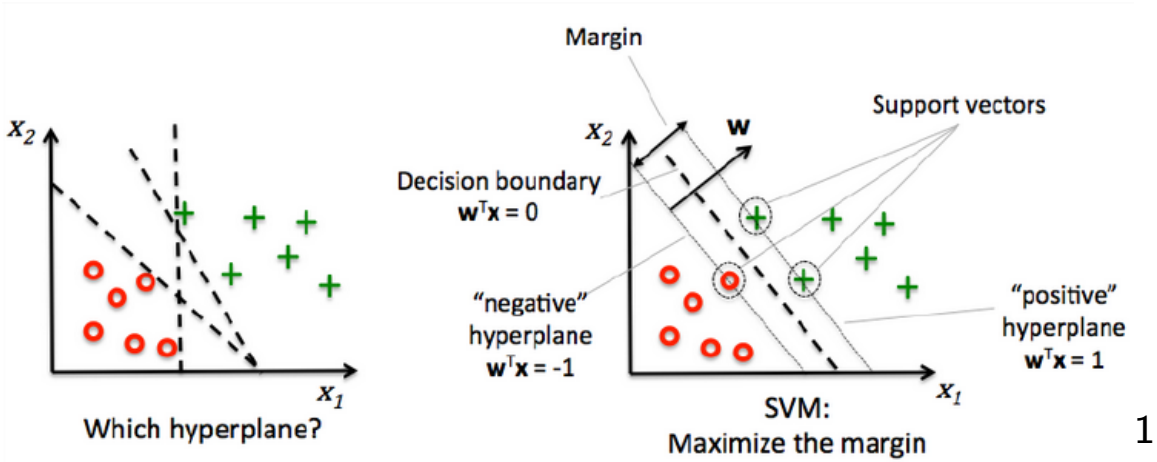

w
$-\frac{b}{\|\mathbf{w}\|}$


w
$-\frac{b}{\|\mathbf{w}\|}$

Perceptron can learn linearly separable functions. Separation plane divides input space in classes $\{0, 1\}$ and is defined by normal vector $\mathbf{w}$ and shift b.



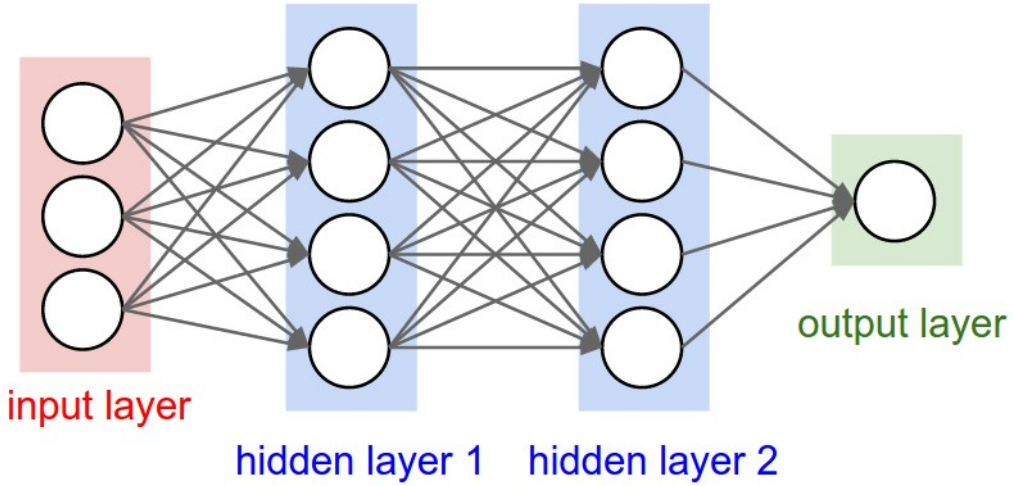
Support vector machine: Perceptron with maximum separation margin.



[1]

[1] From https://en.wikipedia.org/wiki/Activation_function

Multilayer Feedforward Network / Perceptron

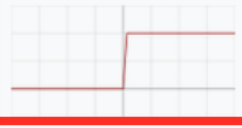
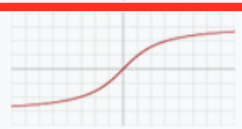
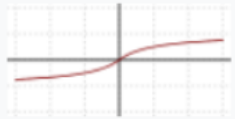


Sequence of linear and nonlinear transforms defined by the recursion:

$$\mathbf{x}^{l+1} = \sigma\left(\mathbf{W}^l \mathbf{x}^l + \mathbf{b}^l\right)$$

- L layers indexed by $l = 1, ..., L$. Input vector $\mathbf{x}^{(0)}$ does not count as a layer.
- $\mathbf{x}^l \in \mathbb{R}^{n_l}$: Activations of n_l neurons at layer l
- Trainable weights $\mathbf{W}^l \in \mathbb{R}^{n_{l-1} \times n_l}$ and biases $\mathbf{b}^l \in \mathbb{R}^{n_l}$ at each layer. W^l_{ij} is connecting neuron j of layer $l-1$ with neuron i of layer l.
- Nonlinear function $\sigma : \mathbb{R} \to \mathbb{R}$. $\sigma(\mathbf{x}) = [\sigma(x_1), ..., \sigma(x_n)]^\top$ is shorthand for applied element-wise application to vector $\mathbf{x} \in \mathbb{R}^n$.
- Output vector: $\hat{\mathbf{y}} = \mathbf{x}^l$ ($\hat{\mathbf{y}}$: network predictions. $\mathbf{y}$: training values)

Common Choices of Nonlinearities

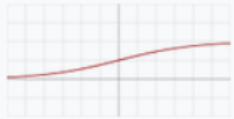
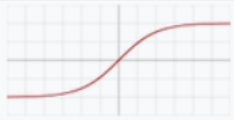
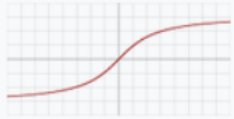
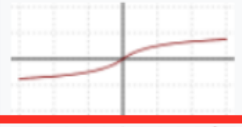
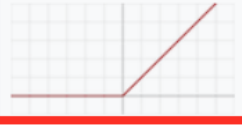
Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$ traditionally used
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$
ArcTan		$f(x) = \tan^{-1}(x)$
Softsign [7][8]		$f(x) = \frac{x}{1+\lvert x\rvert}$
Rectified linear unit (ReLU)[9]		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$

[1]

Drawback: vanishing gradients. For large input weights, activation function saturates and derivative of output vanishes (especially for deep networks).

[1] From https://en.wikipedia.org/wiki/Activation_function

Common Choices of Nonlinearities

Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$
ArcTan		$f(x) = \tan^{-1}(x)$
Softsign [7][8]		$f(x) = \frac{x}{1+\lvert x \rvert}$
Rectified linear unit (ReLU)[9]		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$

Currently most widely used. Empirically easier to train and results in sparse networks. [1]

Non-saturating activation functions – gradients stay finite even for large inputs. Continuously differentiable versions: ELUs and SoftPlus.

[1] From https://en.wikipedia.org/wiki/Activation_function

Universal Representation Theorem (URT)

Sloppy formulation: *A neural network with **a single** hidden layer and a monotonically increasing nonlinearity σ can approximate any continuous function $F : \mathbb{R}^{n_0} \mapsto \mathbb{R}^{n_2}$ from inputs $\mathbb{R}^{n_0}$ to outputs $\mathbb{R}^{n_2}$ with arbitrary accuracy given that sufficiently many hidden neurons n_1 are provided.*

- *Proof for sigmoid activation functions*: G. Cybenko: "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems 2, 303-314 (1989)
- *Generalization*: K. Hornik: "Approximation Capabilities of Multilayer Feedforward Networks", Neural Networks 4, 251-257 (1991)

Caveats:

- Existence of network parameters that approximate F does not mean these parameters can be efficiently found.
- Approximation of a function does not mean exact representation, error might be too large for practical purposes.
- For many complex functions, the number of hidden neurons required to achieve acceptable accuracy is unpractical. $\rightarrow$ deep neural networks.

Universal Representation Theorem (URT)[1]

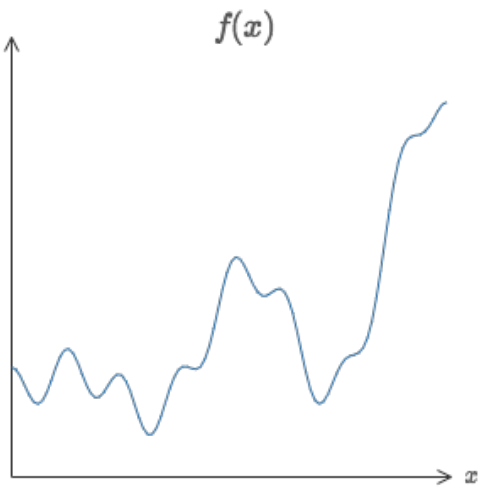
One-dimensional version: Given continuous function $f(x) : \mathbb{R} \mapsto \mathbb{R}$, the URT guarantees that for sufficiently many hidden neurons n_1, there exist weights $(\mathbf{w}^1, \mathbf{b}^1, \mathbf{w}^2, b^2)$, such that the two-layer neural network function

$$\hat{y}(x) = \sigma\left(\left(\mathbf{w}^2\right)^\top \sigma\left(\mathbf{w}^1 x + \mathbf{b}^1\right) + b^2\right)$$

approximates the function $f(x)$ with the desired accuracy ε:

$$|\hat{y}(x) - f(x)| < \varepsilon \quad \forall x \in \mathbb{R}.$$

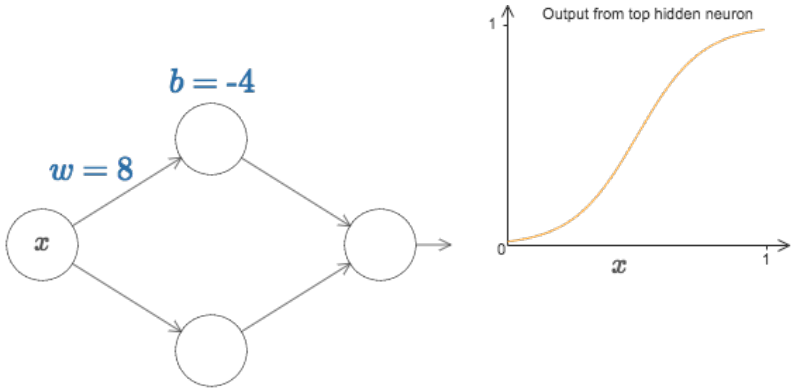
Example: Approximate the following function with $\sigma(z) = (1 + e^{-z})^{-1}$



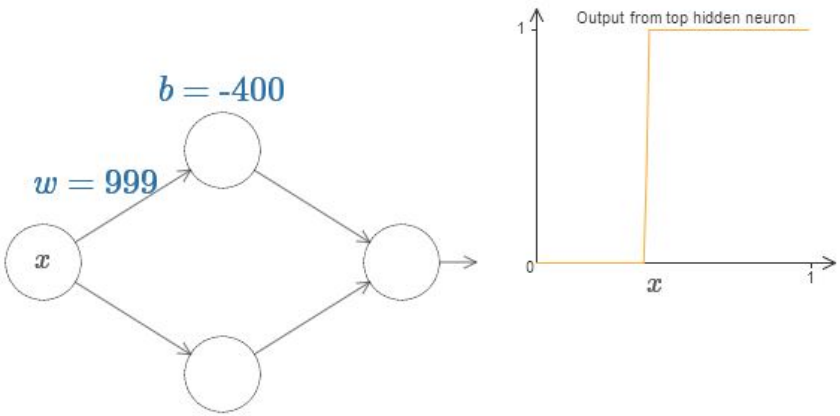
[1] From http://neuralnetworksanddeeplearning.com/chap4.html

Universal Representation Theorem (URT)[1]

Consider output of one hidden neuron.



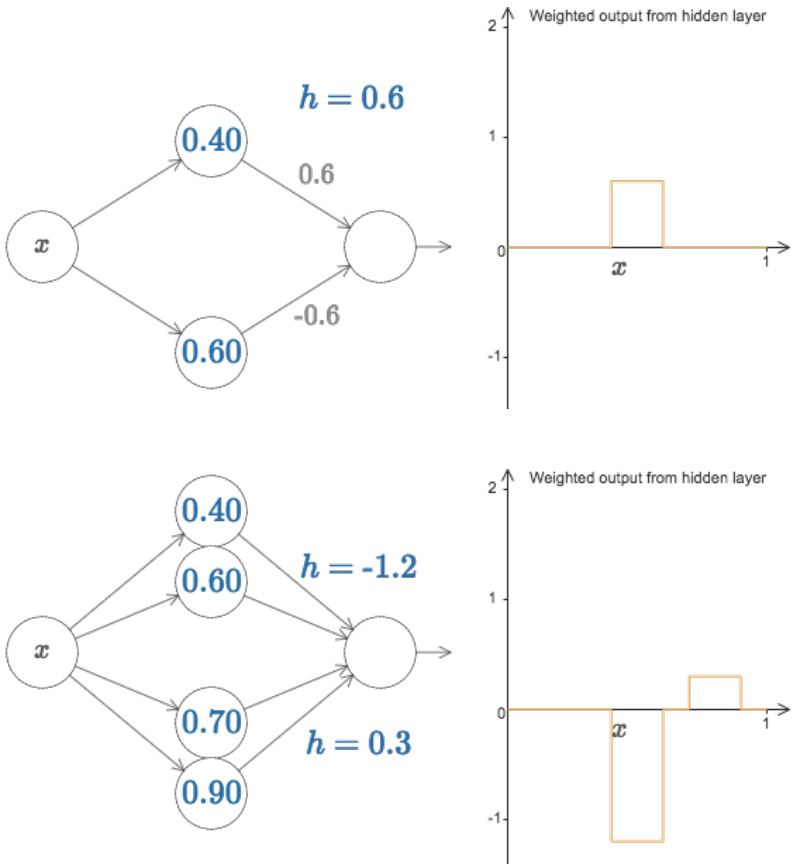
Increasing weight makes the sigmoid more steep, approximating a step function. Use the bias to shift the step where we want it:



[1] From http://neuralnetworksanddeeplearning.com/chap4.html

Universal Representation Theorem (URT)[1]

Characterize each hidden neuron i by the position $s_i = b_i^1 / w_i^1$ where it places the step. As the output neuron performs a weighted sum, we can form arbitrary functions as the input to the output nonlinearity:

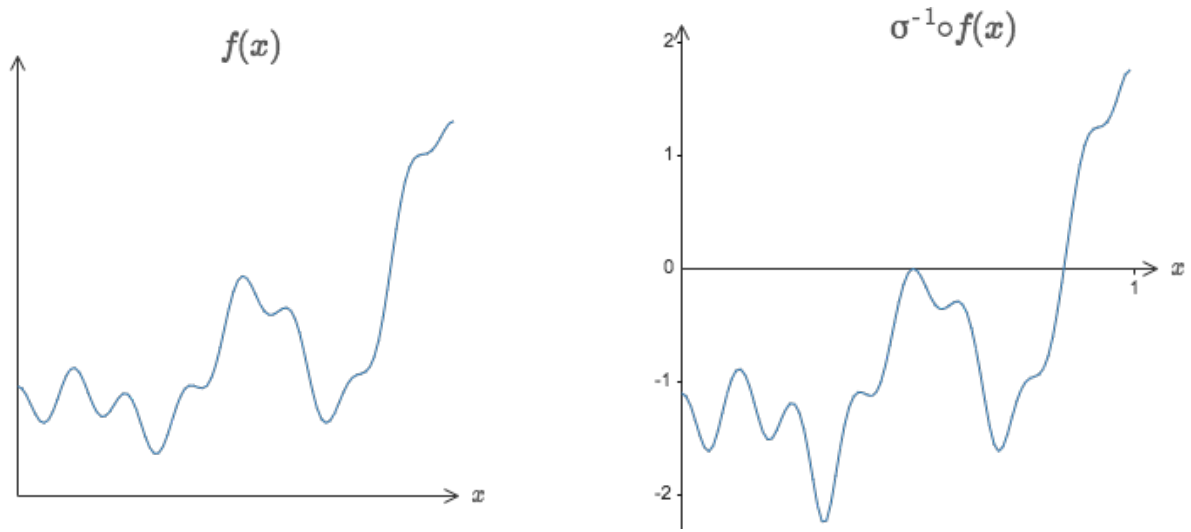


[1] From http://neuralnetworksanddeeplearning.com/chap4.html

Universal Representation Theorem (URT)[1]

In order to deal with the final nonlinearity, we design the weights such that we approximate $\sigma^{-1}(f(x))$ by the input to the output nonlinearity:

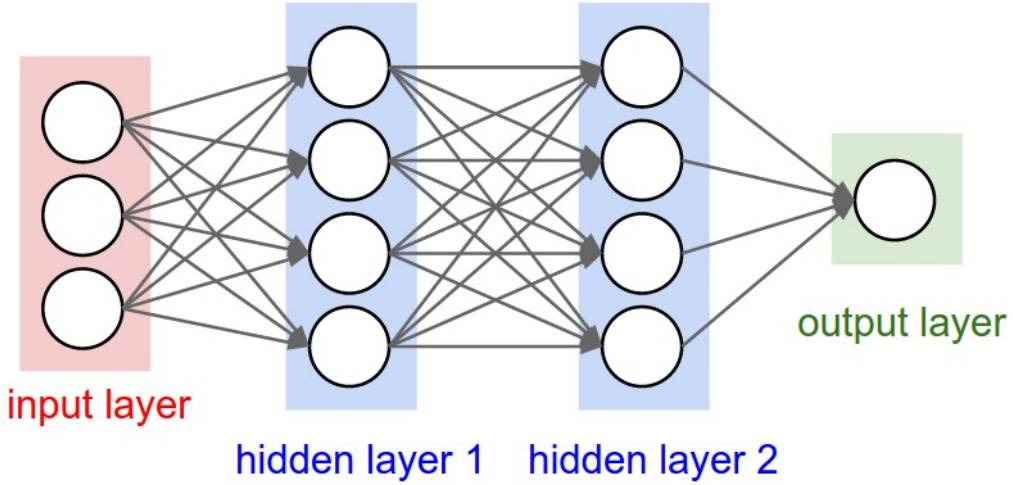
$$f(x) = \sigma\left(\sigma^{-1}(f(x))\right)$$



See http://neuralnetworksanddeeplearning.com/chap4.html for illustration of multidimensional case.

[1] From http://neuralnetworksanddeeplearning.com/chap4.html

Multilayer Feedforward Network



- **Network architecture** is typically the most complex model decision:
  - Number of hidden layers $L-1$ and number of neurons $(n_1, \ldots n_{L-1})$.
  - Type of activation functions σ.
  - Layer types (dense, convolutional, etc. – more details later)
  - Additional regularization terms, e.g. promoting sparsity.
- **Choosing the network architecture**:
  - In principle: Hyperparameter selection problem.
  - In practice: Engineering problem, problem specific architecture.
  - Depends on amount and type of data and available computational resources.
- Intuition: number of parameters should be large enough to prevent underfitting (Advani and Saxe, 2017).

Training Multilayer Neural Networks

Cost or **loss function** C quantifies performance of network with parameters θ to predict observations $\mathbf{X}$.

Learning network weights

$$\hat{\theta} = \arg\min_\theta C(\mathbf{X}, \mathbf{Y}, \theta)$$

Minimimizing *cost* $C \equiv$ maximizing *score* $-C$. Often we just write $C(\theta)$.

Example: for supervised learning with features $\mathbf{X} = (\mathbf{x}_i)_{i=1\ldots N}$ and labels $\mathbf{Y} = (\mathbf{y}_i)_{i=1\ldots N}$. Neural network prediction $\hat{\mathbf{y}}_i(\mathbf{x}_i, \theta)$. Error vector $\mathbf{\Delta}_i = \mathbf{y}_i - \hat{\mathbf{y}}_i(\mathbf{x}_i, \theta)$.

- **Regression**: L2 error. Becomes **mean square error** for $1d$-outputs.

$$C_{L2}(\mathbf{X}, \theta) = \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{\Delta}_i\|_2^2 = \frac{1}{N}\sum_{i=1}^{N} \mathbf{\Delta}_i^\top \mathbf{\Delta}_i$$

- **Classification** to K categories $(s_i)_{i=1,\ldots,N}$. Define **one-hot encoding**:

$$y_{im} = \begin{cases} 1 & \text{if } s_i = m \\ 0 & \text{otherwise,} \end{cases}$$

and use L2 error. More common choice: categorial cross-entropy

Training Multilayer Neural Networks

Neural networks are usually trained with some form of gradient descent:

Simple gradient descent algorithm

1. Initialize θ_0
2. For $t = 0, \ldots, T-1$ or until converged:
   1. Compute gradient $\mathbf{g}(\theta_t) = \nabla_\theta C(\theta_t)$
   2. Update parameters: $\theta_{t+1} = \theta_t - \eta_t \mathbf{g}(\theta_t)$

- **How do we compute the gradient $\nabla_\theta C(\theta_t)$?**
- Networks are functions of functions, so we will need to use the chain rule of differentiation.
- Key is to make differentiation fully automatic so that we can just define the network architecture without worrying about functions and gradients.
  $\rightarrow$ **Backpropagation**.

Backpropagation

- **Notation** (reminder): L Layers indexed by $l = 1, ..., L$. $\mathbf{x}^l \in \mathbb{R}^{n_l}$: Activations of n_l neurons at layer l. Network output: $\hat{\mathbf{y}} = \mathbf{x}^L$.

$$\mathbf{x}^{l+1} = \sigma\left(\mathbf{W}^l \mathbf{x}^l + \mathbf{b}^l\right)$$

  - Trainable weights $\mathbf{W}^l \in \mathbb{R}^{n_l \times n_{l-1}}$ and biases $\mathbf{b}^l \in \mathbb{R}^{n_l}$ at each layer. W^l_{ij} is connecting neuron j of layer $l-1$ with neuron i of layer l.
  - Nonlinear activation functions $\sigma : \mathbb{R} \to \mathbb{R}$, applied element-wise.

- Activation of a single neuron:

$$x^l_i = \sigma\left(z^l_i\right)$$
$$z^l_i = \sum_j w^l_{ij} x^{l-1}_j + b^l_i$$

Backpropagation

- Loss gradient with respect to the activations of the output neurons:

$$\frac{\partial C}{\partial \hat{y}_i} = \frac{\partial C}{\partial x_i^L}$$

- Depends on the definition of the loss function. For mean square error:

$$C(\mathbf{X}, \theta) = \frac{1}{N} \sum_{k=1}^{N} c(\mathbf{x}_k, y_k, \theta) \qquad \frac{\partial C(\mathbf{X}, \theta)}{\partial \hat{y}_i} = \frac{1}{N} \sum_{k=1}^{N} \frac{\partial c(\mathbf{x}_k, y_k, \theta)}{\partial \hat{y}_i}$$

with

$$c(\mathbf{x}, y, \theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i(\mathbf{x}, \theta))^2 \qquad \frac{\partial c(\mathbf{x}, y, \theta)}{\partial \hat{y}_i} = 2(\hat{y}_i(\mathbf{x}, \theta) - y_i)$$

Loss function needs to be differentiable.

- Loss-specific derivatives can be implemented with the loss functions in a neural network package.

Backpropagation

- **Error**: Loss gradient with respect to the ith weighted input

$$\mathbf{e}_i^L = \frac{\partial C}{\partial z_i^L} = \underbrace{\frac{\partial c(\mathbf{x}, y, \theta)}{\partial \hat{y}_i}}_{\text{loss derivative}} \frac{\partial \hat{y}_i}{\partial z_i^L} = \frac{\partial c(\mathbf{x}, y, \theta)}{\partial \sigma(z_i^L)} \frac{\partial \sigma(z_i^L)}{\partial z_i^L} = \frac{\partial c(\mathbf{x}, y, \theta)}{\partial \sigma(z_i^L)} \underbrace{\sigma'(z_i^L)}_{\text{activation derivati}}$$

  The derivative of the fixed activation function can be implemented along with that function, e.g.:

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

- **Backpropagate error** to earlier layers using $z_j^{l+1} = \sum_k w_{jk}^{l+1} x_k^l + b_j^{l+1}$:

$$\begin{aligned}
\mathbf{e}_i^l = \frac{\partial C}{\partial z_i^l} &= \sum_j \frac{\partial C}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \mathbf{e}_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_i^l} \\
&= \sum_j \mathbf{e}_j^{l+1} \frac{\partial}{\partial z_i^l} \left(\sum_k w_{jk}^{l+1} \sigma(z_k^l) + b_j^{l+1} \right) \\
&= \sigma'(z_i^l) \sum_j \mathbf{e}_j^{l+1} w_{ji}^{l+1}
\end{aligned}$$

Backpropagation

- **Compute gradients** at each layer using $z_i^l = \sum_k w_{ik}^l x_k^{l-1} + b_i^l$:

$$\frac{\partial C}{\partial b_i^l} = \frac{\partial C}{\partial z_i^l}\frac{\partial z_i^l}{\partial b_i^l} = \frac{\partial C}{\partial z_i^l} = \mathbf{e}_i^l$$

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C}{\partial z_i^l}\frac{\partial z_i^l}{\partial w_{ij}^l} = \frac{\partial C}{\partial z_i^l} x_j^{l-1} = \mathbf{e}_i^l x_j^{l-1}$$

- **Weight update**:

$$b_i^l \leftarrow b_i^l - \eta \mathbf{e}_i^l$$

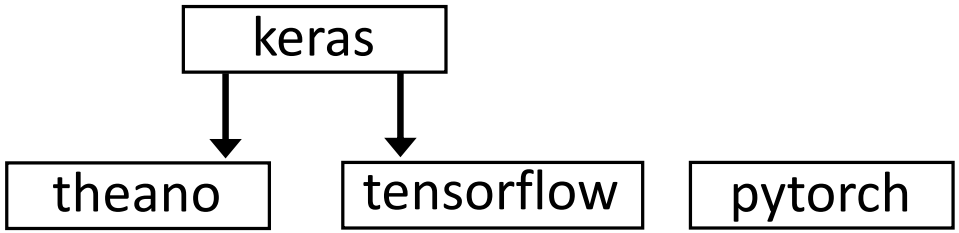
$$w_{ij}^l \leftarrow w_{ij}^l - \eta \mathbf{e}_i^l x_j^{l-1}$$

with learning rate η.

Forward- and Backpropagation

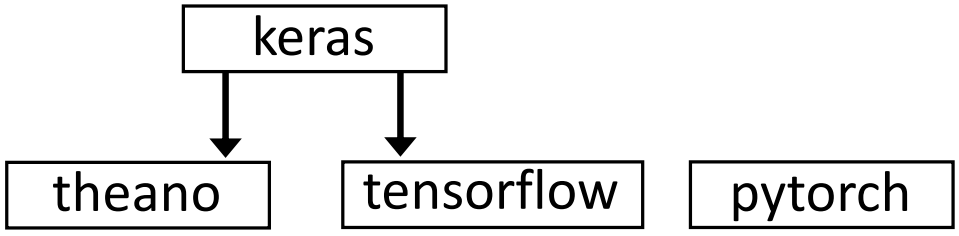
1. **Activation at input layer**: Input activations x_i^0.
2. **Feedforward**: For each layer, compute weighted sum and nonlinear transformation, and thus activations of neurons x_i^l
3. **Error at output layer**: Calculate the error of output layer using the derivatives of the cost function.
4. **Backpropagate the error**: Propagate the error backwards and compute Δ_i^l for all layers.
5. **Calculate gradients**: $\frac{dC}{db_i^l}$ and $\frac{dC}{dw_{ij}^l}$ of cost function with respect to all trainble weights.
   Neural network packages (Theano, TensorFlow, PyTorch, Autograd, ...) have gradients defined for all elementary functions. This permits automatic differentiation using the product rule for composite functions.
6. **Train**: Change weights b_i^l and w_{ij}^l in the direction opposite gradient.

Software



- **keras**: high-level python package.
  - Easy to learn, but limited flexibility
  - Runs on CPUs (slow) and NVidia GPUs (via theano, tensorflow)
  - Recommended installation: direct pip install of the newest compatible keras and tensorflow packages.
- **theano, tensorflow**: low-level python packages for GPU-accelerated tensor operations.
  - Harder to learn, maximum flexibility and speed.
  - Mathematical code defines a graph with operations as nodes and tensors passed along the edges.
  - Before executing, GPU code for graph is generated and compiled.
  - Derivatives are defined for basic operators, enabling automatic backprop for most code.
  - Recommended installation: pip or docker.

Software



- **pytorch**: simplified low-level python package for GPU-accelerated tensor operations.
  - Learning curve between keras and tensorflow.
  - Allows dynamic changes in network structures.
  - More intuitive than tensorflow because operations are executed on demand (dynamic linking of precompiled modules instead of code generation)
  - Recommended installation: conda

Steps in neural network training

1. **Data preparation**: collect data, featurize and preprocess it:
   1. Benchmark datasets can be downloaded via NN packages (e.g. MNIST, CIFAR)
   2. Featurization means to define a vector or tensor representation of the data which can be processed by NN packages.
   3. Splitting into training and test data (size of test set should be related to complexity of learning task - e.g. 80/20 for MNIST with 10 classes, 50/50 for ImageNet with 1000 classes)
   4. Shuffle data sets.
   5. Center data: subtract mean over all samples from each sample.
   6. Rescale data: when approximately normal, divide the standard deviation. Otherwise, rescaled by the maximum absolute value to move data to interval $[-1, 1]$. Rescaling ensures that the weights of the DNN are of a similar order of magnitude.
   7. Data augmentation, i.e. distorting data samples from the existing dataset in some way to enhance size the dataset, e.g. to exploit invariances.

Steps in neural network training

1. **Data preparation**: collect data, featurize and preprocess it:
2. **Network model**: Define layers and connectivity, i.e. architecture of the network
3. **Loss function**: Define or implement the cost / loss function that should be minimized.
4. **Optimizer**: Choose the Optimizer (e.g. ADAM, RMSprop)
5. **Train model**: Feed data into (model / loss / optimizer) structure in order to train it, often in minibatches.
6. **Evaluate model**: Track the convergence of training and test loss and evaluate performance on training and test data.
7. **Adjust hyperparameters / architecture** and repeat from 2 if necessary. Since learning is expensive, often useful to choose a data subset for exploration.

Key advances in deep network training

Until 2009, deep networks were rarely used as training them seemed too difficult. Key advances that enabled the training of deep networks include:

1. **Rectifiers**: Changing from activation functions such as logistic, tanh or arctan to rectifiers (ReLu, ELU, SoftPlus) avoids the vanishing gradient problem, makes the loss surface less "frustrated" and thus improves convergence.
2. **Stochastic Gradient Descent**: Changing from full gradient descent to stochastic gradient descent resulted in significant reduction of the computational cost to convergence (cheaper iterations and ability to escape flat local minima and saddle points)
3. **GPU implementations** of neural networks (Theano, TensorFlow), the resulting computational speedup, and the flexibility of automatic differentiation.

Data versus learning performance