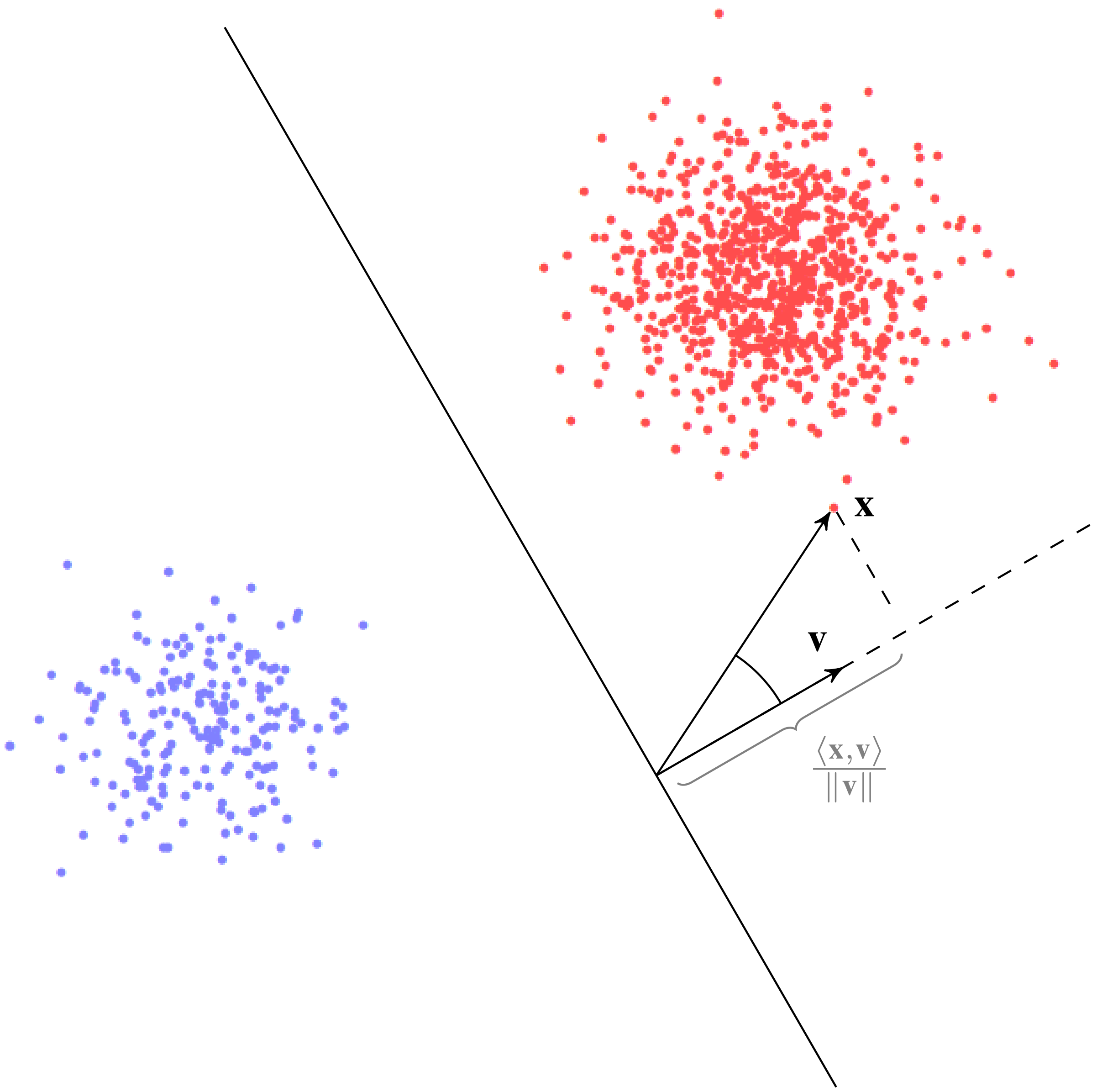
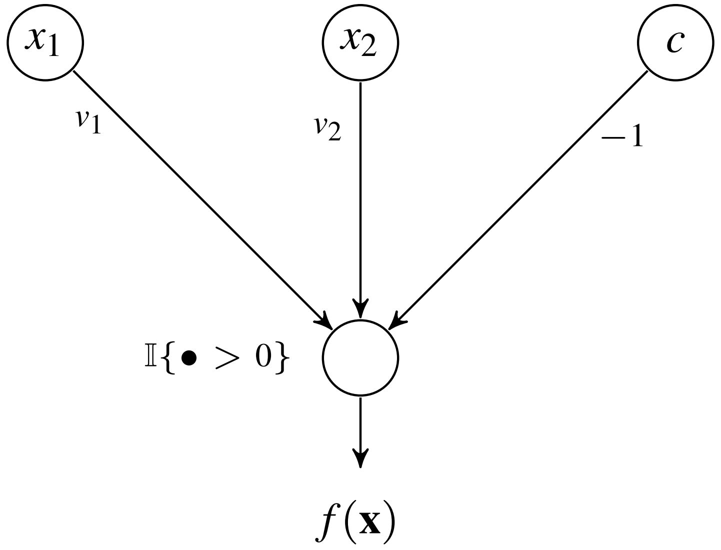
Recall: Linear Classification



$\mathbf{x}$
$\mathbf{v}$
$\frac{\langle \mathbf{x}, \mathbf{v} \rangle}{\|\mathbf{v}\|}$

$$f(\mathbf{x}) = \operatorname{sgn}(\langle \mathbf{v}, \mathbf{x} \rangle - c)$$

Linear Classifier in $\mathbb{R}^2$ as Two-Layer NN



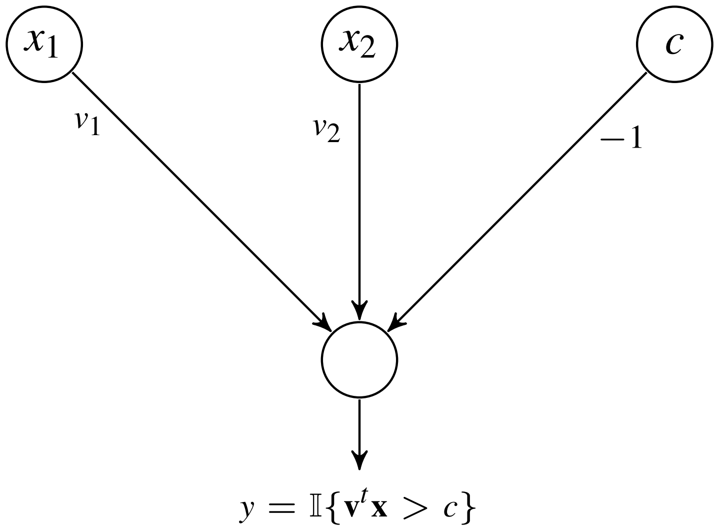
$$f(\mathbf{x}) \quad = \quad \mathbb{I}\{\ v_1x_1 + v_2x_2 + v_3x_3 + (-1)c\ >\ 0\ \} \quad = \quad \mathbb{I}\{\langle \mathbf{v}, \mathbf{x}\rangle > c\}$$

Equivalent to linear classifier

The linear classifier on the previous slide and f differ only in whether they encode the “blue” class as -1 or as 0:

$$\operatorname{sgn}(\langle \mathbf{v}, \mathbf{x}\rangle - c)\ =\ 2f(\mathbf{x}) - 1$$

REMARKS

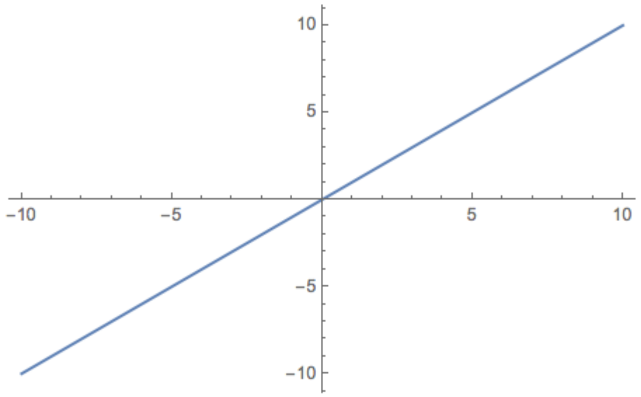


- This neural network represents a linear two-class classifier (on $\mathbb{R}^2$).
- We can more generally define a classifier on $\mathbb{R}^d$ by adding input units, one per dimension.
- It does not specify the training method.
- To train the classifier, we need a cost function and an optimization method.

Typical Component Functions

Linear units

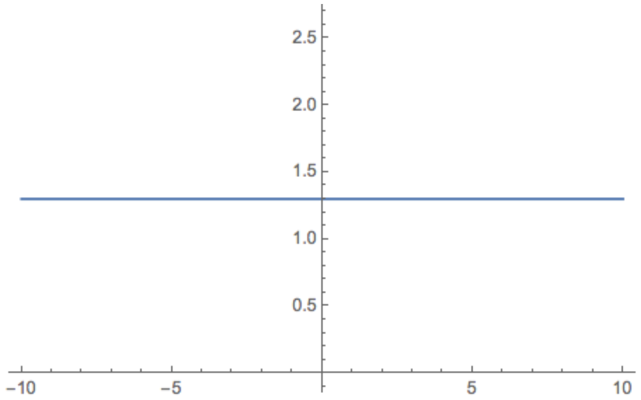
$$\phi(x) = x$$



This function simply “passes on” its incoming signal. These are used for example to represent inputs (data values).

Constant functions

$$\phi(x) = c$$

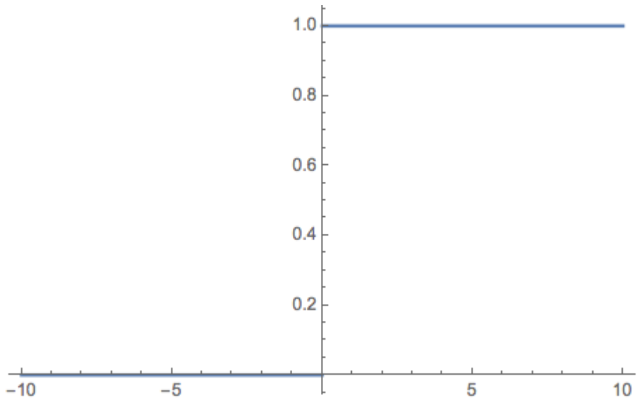


These can be used e.g. in combination with an indicator function to define a threshold, as in the linear classifier above.

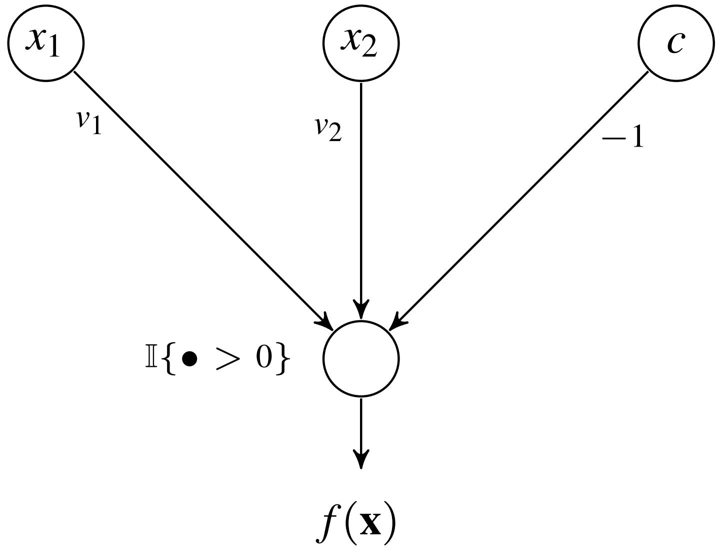
TYPICAL COMPONENT FUNCTIONS

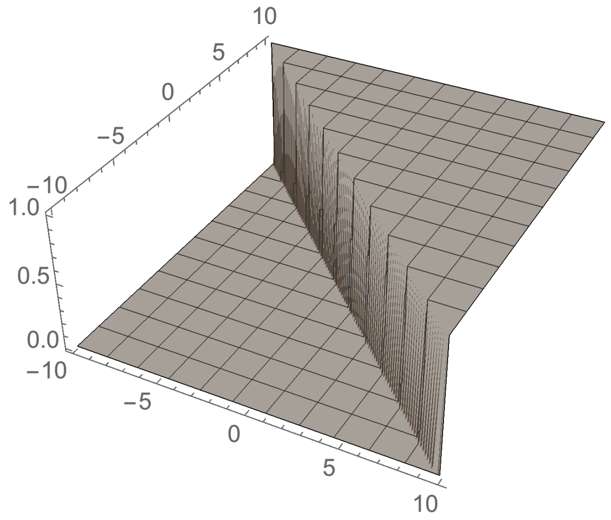
Indicator function

$$\phi(x) = \mathbb{I}\{x > 0\}$$



Example: Final unit is indicator

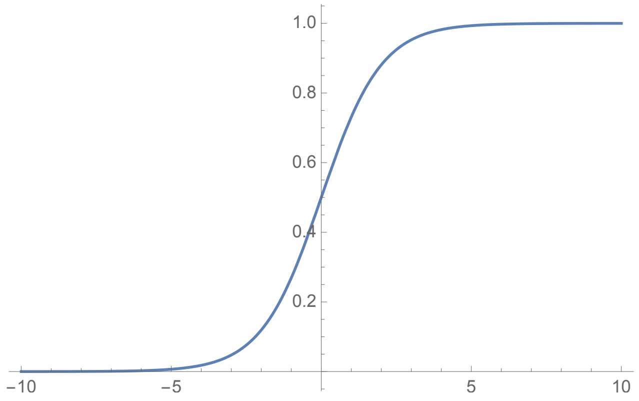




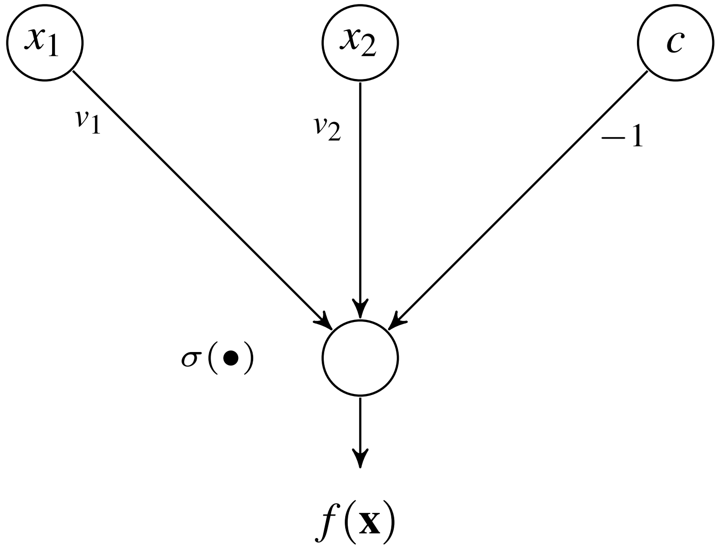
Typical Component Functions

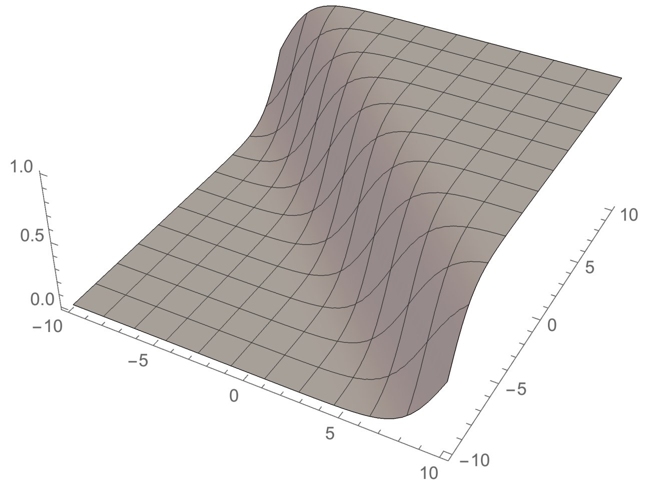
Sigmoids

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



Example: Final unit is sigmoid

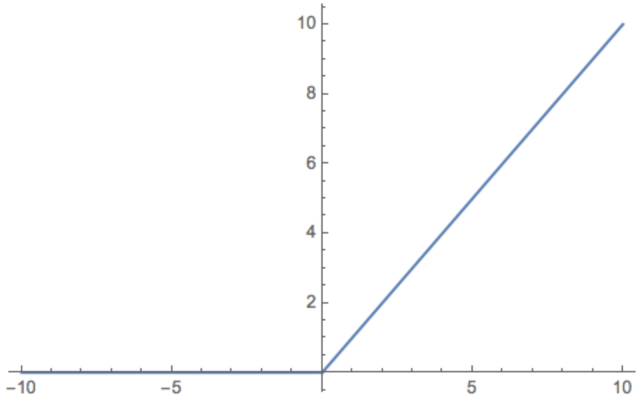




TYPICAL COMPONENT FUNCTIONS

Rectified linear units

$$\phi(x) = \max\{0, x\}$$



These are currently perhaps the most commonly used unit in the “inner” layers of a neural network (those layers that are not the input or output layer).

Hidden Layers and Nonlinear Functions

Hidden units

- Any nodes (or “units”) in the network that are neither input nor output nodes are called **hidden**.
- Every network has an input layer and an output layer.
- If there any additional layers (which hence consist of hidden units), they are called **hidden layers**.

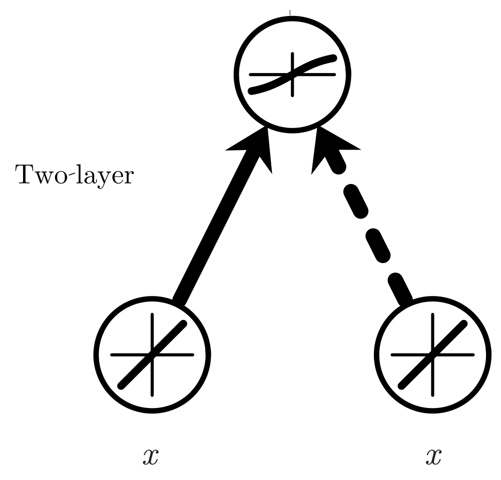
Linear and nonlinear networks

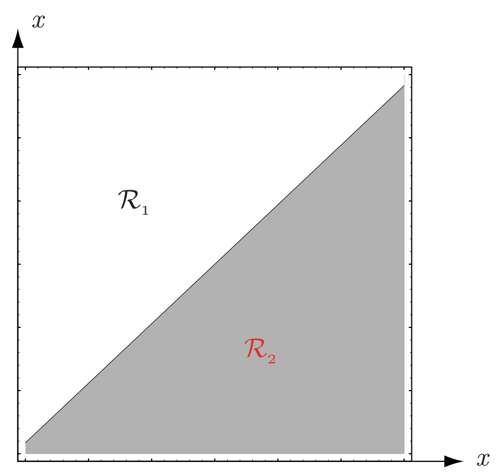
- If a network has no hidden units, then

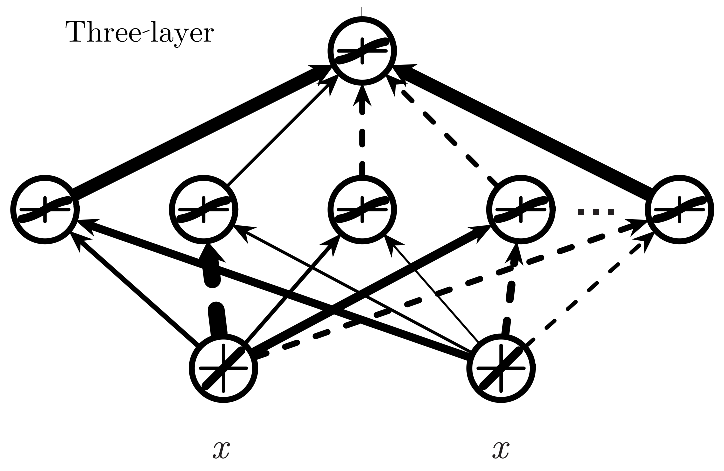
  $$f_i(\mathbf{x}) = \phi_i(\langle \mathbf{w}_i, \mathbf{x} \rangle)$$

  That means: f is a linear functions, except perhaps for the final application of ϕ.
- For example: In a classification problem, a two layer network can only represent linear decision boundaries.
- Networks with at least one hidden layer can represent nonlinear decision surfaces.

Two vs Three Layers







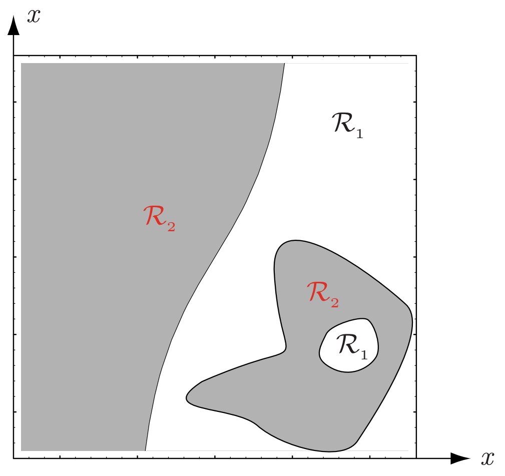
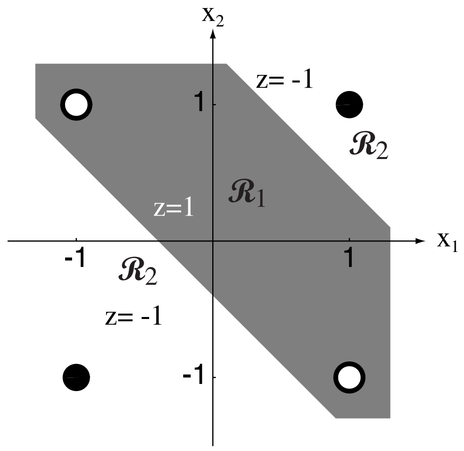
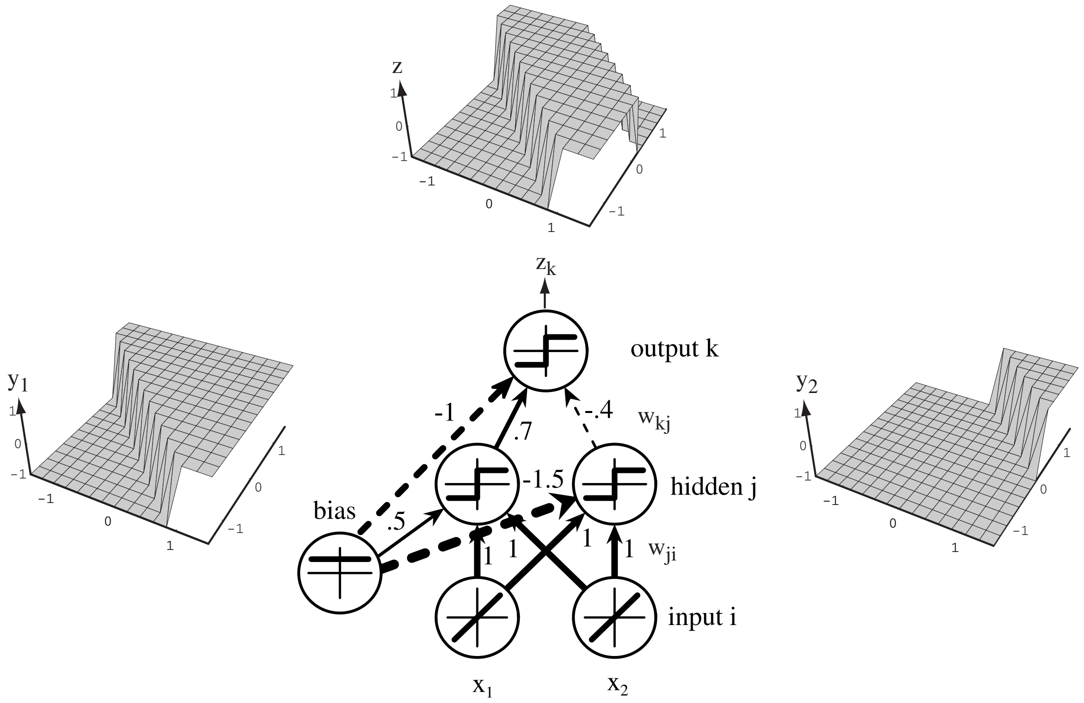


Illustration: R.O. Duda, P.E. Hart, D.G. Stork, *Pattern Classification*, Wiley 2001

The XOR Problem



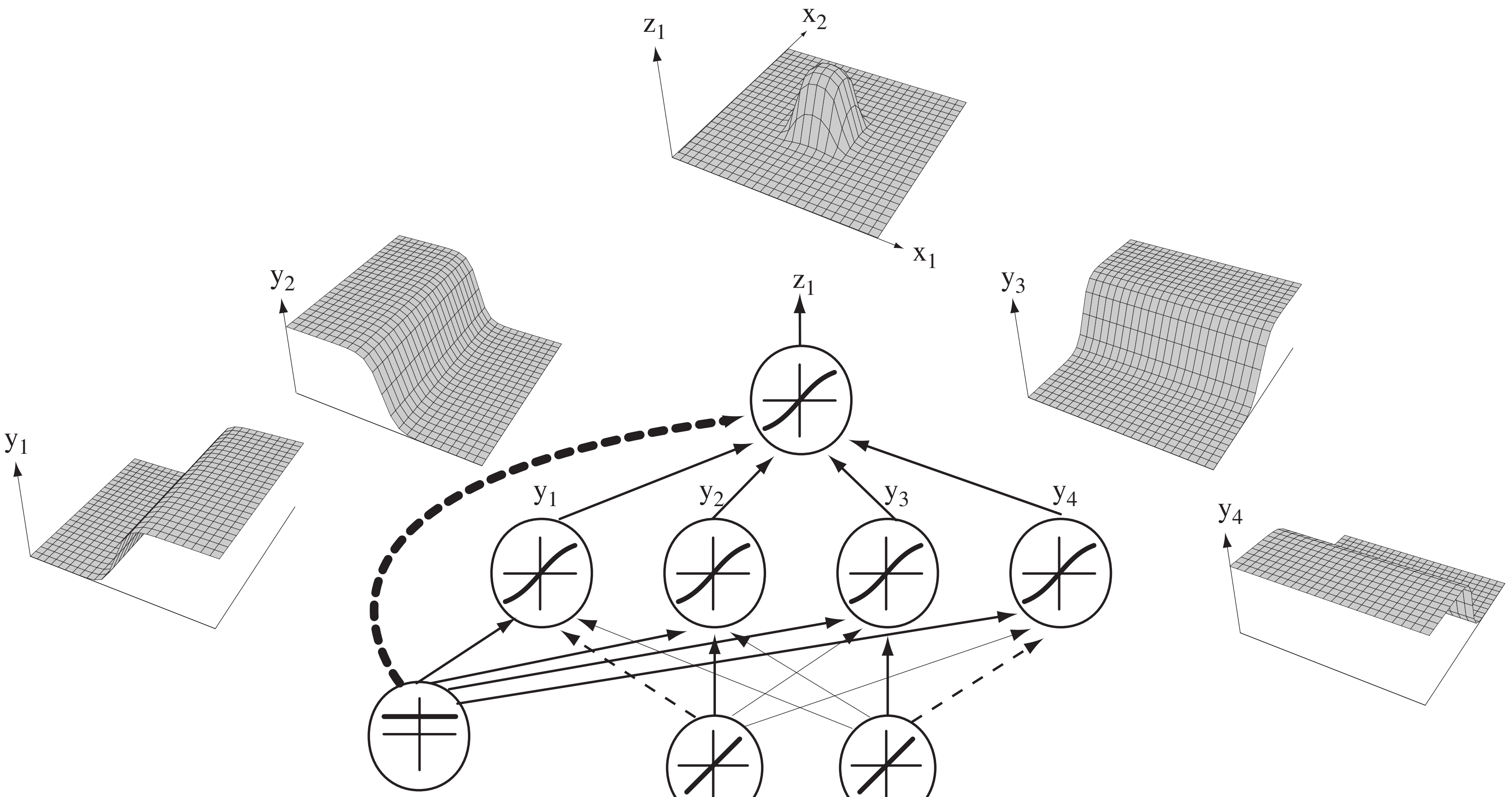
Solution regions we would like to represent



Neural network representation

- Two ridges at different locations are substracted from each other.
- That generates a region bounded on both sides.
- A linear classifier cannot represent this decision region.
- Note this requires at least one hidden layer.

Illustration: R.O. Duda, P.E. Hart, D.G. Stork, *Pattern Classification*, Wiley 2001


z1
x2
x1
y2
y3
y1
y4
z1
y1
y2
y3
y4

NUMBER OF LAYERS

We have observed

- We have seen that two-layer classification networks always represent linear class boundaries.
- With three layers, the boundaries can be non-linear.

Obvious question

- What happens if we use more than three layers? Do four layers again increase expressive power?

WIDTH VS DEPTH

A neural network represents a (typically) complicated function f by simple functions $\phi_i^{(k)}$.

What functions can be represented?

A well-known result in approximation theory says: Every continuous function $f : [0, 1]^d \to \mathbb{R}$ can be represented in the form

$$f(\mathbf{x}) = \sum_{j=1}^{2d+1} \xi_j \Big(\sum_{i=1}^{d} \tau_{ij}(x_i) \Big)$$

where ξ_i and τ_{ij} are functions $\mathbb{R} \to \mathbb{R}$. A similar result shows one can approximate f to arbitrary precision using specifically sigmoids, as

$$f(\mathbf{x}) \approx \sum_{j=1}^{M} w_j^{(2)} \sigma \Big(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + c_i \Big)$$

for some finite M and constants c_i.

Note the representations above can both be written as neural networks with three layers (i.e. with one hidden layer).

Not examinable.

WIDTH VS DEPTH

Depth rather than width

- The representations above can achieve arbitrary precision with a single hidden layer (roughly: a three-layer neural network can represent any continuous function).
- In the first representation, ξ_j and τ_{ij} are "simpler" than f because they map $\mathbb{R} \to \mathbb{R}$.
- In the second representation, the functions are more specific (sigmoids), and we typically need more of them (M is large).
- That means: The price of precision are many hidden units, i.e. the network grows wide.
- The last years have shown: We can obtain very good results by limiting layer width, and instead increasing depth (= number of layers).
- There is no coherent theory yet to properly explain this behavior.

Limiting width

- Limiting layer width means we limit the degrees of freedom of each function $f^{(k)}$.
- That is a notion of parsimony.
- Again: There seem to be a lot of interesting questions to study here, but so far, we have no real answers.

Not examinable.

Training Neural Networks

Task

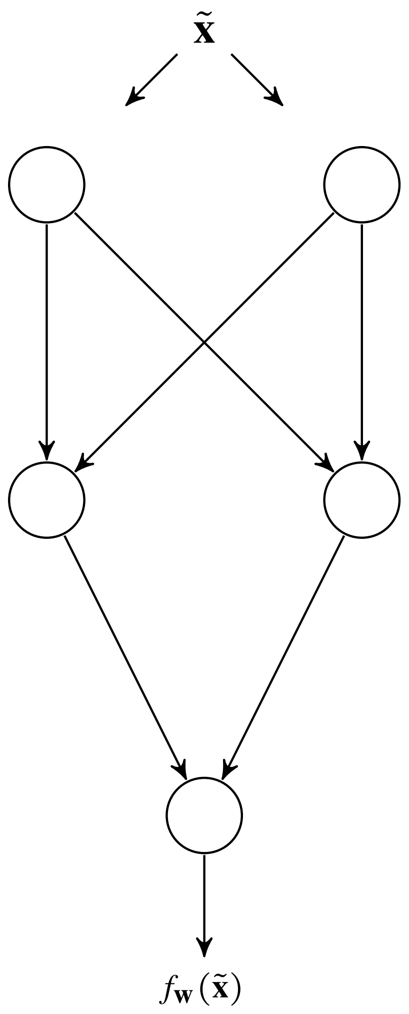
- We decide on a neural network "architecture": We fix the network diagram, including all functions ϕ at the units. Only the weights w on the edges can be changed during by training algorithm. Suppose the architecture we choose has d_1 input units and d_2 output units.
- We collect all weights into a vector $\mathbf{w}$. The entire network then represents a function $f_{\mathbf{w}}(\mathbf{x})$ that maps $\mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$.
- To "train" the network now means that, given training data, we have to determine a suitable parameter vector $\mathbf{w}$, i.e. we fit the network to data by fitting the weights.

More specifically: Classification

Suppose the network is meant to represent a two-class classifier.

- That means the output dimension is $d_2 = 1$, so $f_{\mathbf{w}}$ is a function $\mathbb{R}^{d_1} \to \mathbb{R}$.
- We are given data $\mathbf{x}_1, \mathbf{x}_2, \ldots$ with labels $y_1, y_2, \ldots$.
- We split this data into training, validation and test data, according to the requirements of the problem we are trying to solve.
- We then fit the network to the training data.

Training Neural Networks



- We run each training data point $\tilde{\mathbf{x}}_i$ through the network $f_{\mathbf{w}}$ and compare $f_{\mathbf{w}}(\tilde{\mathbf{x}}_i)$ to $\tilde{y}_i$ to measure the error.
- Recall how gradient descent works: We make "small" changes to $\mathbf{w}$, and choose the one which decreases the error most. That is one step of the gradient scheme.
- For each such changed value $\mathbf{w}'$, we again run each training data point $\tilde{\mathbf{x}}_i$ through the network $f_{\mathbf{w}'}$, and measure the error by comparing $f_{\mathbf{w}'}(\tilde{\mathbf{x}}_i)$ to $\tilde{y}_i$.

TRAINING NEURAL NETWORKS

Error measure

- We have to specify how we compare the network's output $f_{\mathbf{w}}(\mathbf{x})$ to the correct answer y.
- To do so, we specify a function D with two arguments that serves as an error measure.
- The choice of D depends on the problem.

Typical error measures

- Classification problem:

$$D(\hat{y}, y) := y \log \hat{y} \qquad \text{(with convention } 0 \log 0 = 0\text{)}$$

- Regression problem:

$$D(\hat{y}, y) := \|y - \hat{y}\|^2$$

Training as an optimization problem

- Given: Training data $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ with labels y_i.
- We specify an error measure D, and define the total error on the training set as

$$J(\mathbf{w}) := \sum_{i=1}^{n} D(f_{\mathbf{w}}(\tilde{\mathbf{x}}_i), \tilde{y}_i)$$

BACKPROPAGATION

Training problem

In summary, neural network training attempts to solve the optimization problem

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w})$$

using gradient descent. For feed-forward networks, the gradient descent algorithm takes a specific form that is called *backpropagation*.

Backpropagation is gradient descent applied to $J(\mathbf{w})$ in a feed-forward network.

In practice: Stochastic gradient descent

- The vector $\mathbf{w}$ can be very high-dimensional. In high dimensions, computing a gradient is computationally expensive, because we have to make "small changes" to $\mathbf{w}$ in many different directions and compare them to each other.
- Each time the gradient algorithm computes $J(\mathbf{w}')$ for a changed value $\mathbf{w}'$, we have to apply the network to every data point, since $J(\mathbf{w}') = \sum_{i=1}^{n} D(f_{\mathbf{w}'}(\tilde{\mathbf{x}}_i), \tilde{y}_i)$.
- To save computation, the gradient algorithm typically computes $D(f_{\mathbf{w}'}(\tilde{\mathbf{x}}_i), \tilde{y}_i)$ only for some small subset of a the training data. This subset is called a *mini batch*, and the resulting algorithm is called **stochastic gradient descent**.

Backpropagation

Neural network training optimization problem

$$\min_{\mathbf{w}} J(\mathbf{w})$$

The application of gradient descent to this problem is called *backpropagation*.

Backpropagation is gradient descent applied to $J(\mathbf{w})$ in a feed-forward network.

Deriving backpropagation

- We have to evaluate the derivative $\nabla_{\mathbf{w}} J(\mathbf{w})$.
- Since J is additive over training points, $J(\mathbf{w}) = \sum_n J_n(\mathbf{w})$, it suffices to derive $\nabla_{\mathbf{w}} J_n(\mathbf{w})$.

The next few slides were written for a different class, and you are not expected to know their content. I show them only to illustrate the interesting way in which gradient descent interleaves with the feed-forward architecture.

BACKPROPAGATION

Deriving backpropagation

- We have to evaluate the derivative $\nabla_{\mathbf{w}} J(\mathbf{w})$.
- Since J is additive over training points, $J(\mathbf{w}) = \sum_n J_n(\mathbf{w})$, it suffices to derive $\nabla_{\mathbf{w}} J_n(\mathbf{w})$.

Not examinable.

Chain Rule

Recall from calculus: Chain rule

Consider a composition of functions $f \circ g(x) = f(g(x))$.

$$\frac{d(f \circ g)}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

If the derivatives of f and g are f' and g', that means: $\frac{d(f \circ g)}{dx}(x) = f'(g(x))g'(x)$

Application to feed-forward network

Let $\mathbf{w}^{(k)}$ denote the weights in layer k. The function represented by the network is

$$f_{\mathbf{w}}(\mathbf{x}) \;=\; f_{\mathbf{w}}^{(K)} \circ \cdots \circ f_{\mathbf{w}}^{(1)}(\mathbf{x}) \;=\; f_{\mathbf{w}^{(K)}}^{(K)} \circ \cdots \circ f_{\mathbf{w}^{(1)}}^{(1)}(\mathbf{x})$$

To solve the optimization problem, we have to compute derivatives of the form

$$\frac{d}{d\mathbf{w}} D(f_{\mathbf{w}}(\mathbf{x}_n), y_n) = \frac{dD(\,\bullet\,, y_n)}{df_{\mathbf{w}}}\frac{df_{\mathbf{w}}}{d\mathbf{w}}$$

Not examinable.

Decomposing the Derivatives

- The chain rule means we compute the derivates layer by layer.
- Suppose we are only interested in the weights of layer k, and keep all other weights fixed. The function f represented by the network is then

$$f_{\mathbf{w}^{(k)}}(\mathbf{x}) \;=\; f^{(K)} \circ \cdots \circ f^{(k+1)} \circ f^{(k)}_{\mathbf{w}^{(k)}} \circ f^{(k-1)} \circ \cdots \circ f^{(1)}(\mathbf{x})$$

- The first $k-1$ layers enter only as the function value of $\mathbf{x}$, so we define

$$\mathbf{z}^{(k)} := f^{(k-1)} \circ \cdots \circ f^{(1)}(\mathbf{x})$$

  and get

$$f_{\mathbf{w}^{(k)}}(\mathbf{x}) \;=\; f^{(K)} \circ \cdots \circ f^{(k+1)} \circ f^{(k)}_{\mathbf{w}^{(k)}}(\mathbf{z}^{(k)})$$

- If we differentiate with respect to $\mathbf{w}^{(k)}$, the chain rule gives

$$\frac{d}{d\mathbf{w}^{(k)}} f_{\mathbf{w}^{(k)}}(\mathbf{x}) \;=\; \frac{df^{(K)}}{df^{(K-1)}} \cdots \frac{df^{(k+1)}}{df^{(k)}} \cdot \frac{df^{(k)}_{\mathbf{w}^{(k)}}}{d\mathbf{w}^{(k)}}$$

Not examinable.

Within a Single Layer

- Each $f^{(k)}$ is a vector-valued function $f^{(k)} : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}}$.
- It is parametrized by the weights $\mathbf{w}^{(k)}$ of the kth layer and takes an input vector $\mathbf{z} \in \mathbb{R}^{d_k}$.
- We write $f^{(k)}(\mathbf{z}, \mathbf{w}^{(k)})$.

Layer-wise derivative

Since $f^{(k)}$ and $f^{(k-1)}$ are vector-valued, we get a Jacobian matrix

$$\frac{df^{(k+1)}}{df^{(k)}} = \begin{pmatrix} \frac{\partial f_1^{(k+1)}}{\partial f_1^{(k)}} & \cdots & \frac{\partial f_1^{(k+1)}}{\partial f_{d_k}^{(k)}} \\ \vdots & & \vdots \\ \frac{\partial f_{d_{k+1}}^{(k+1)}}{\partial f_1^{(k)}} & \cdots & \frac{\partial f_{d_{k+1}}^{(k+1)}}{\partial f_{d_k}^{(k)}} \end{pmatrix} =: \Delta^{(k)}(\mathbf{z}, \mathbf{w}^{(k+1)})$$

- $\Delta^{(k)}$ is a matrix of size $d_{k+1} \times d_k$.
- The derivatives in the matrix quantify how $f^{(k+1)}$ reacts to changes in the argument of $f^{(k)}$ if the weights $\mathbf{w}^{(k+1)}$ and $\mathbf{w}^{(k)}$ of both functions are fixed.

Not examinable.

Backpropagation algorithm

Let $\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(K)}$ be the current settings of the layer weights. These have either been computed in the previous iteration, or (in the first iteration) are initialized at random.

Step 1: Forward pass

We start with an input vector $\mathbf{x}$ and compute

$$\mathbf{z}^{(k)} := f^{(k)} \circ \cdots \circ f^{(1)}(\mathbf{x})$$

for all layers k.

Step 2: Backward pass

- Start with the last layer. Update the weights $\mathbf{w}^{(K)}$ by performing a gradient step on

  $$D\big(f^{(K)}(\mathbf{z}^{(K)}, \mathbf{w}^{(K)}), y\big)$$

  regarded as a function of $\mathbf{w}^{(K)}$ (so $\mathbf{z}^{(K)}$ and y are fixed). Denote the updated weights $\tilde{\mathbf{w}}^{(K)}$.
- Move backwards one layer at a time. At layer k, we have already computed updates $\tilde{\mathbf{w}}^{(K)}, \dots, \tilde{\mathbf{w}}^{(k+1)}$. Update $\mathbf{w}^{(k)}$ by a gradient step, where the derivative is computed as

  $$\Delta^{(K-1)}(\mathbf{z}^{(K-1)}, \tilde{\mathbf{w}}^{(K)}) \cdot \ldots \cdot \Delta^{(k)}(\mathbf{z}^{(k)}, \tilde{\mathbf{w}}^{(k+1)}) \frac{df^{(k)}}{d\mathbf{w}^{(k)}}(\mathbf{z}, \mathbf{w}^{(k)})$$

On reaching level 1, go back to step 1 and recompute the $\mathbf{z}^{(k)}$ using the updated weights.

Not examinable.

SUMMARY: BACKPROPAGATION

- Backpropagation is a gradient descent method for the optimization problem

$$\min_{\mathbf{w}} J(\mathbf{w}) = \sum_{i=1}^{N} D(f_{\mathbf{w}}(\mathbf{x}_i), y_i)$$

  D must be chosen such that it is additive over data points.

- It alternates between forward passes that update the layer-wise function values $\mathbf{z}^{(k)}$ given the current weights, and backward passes that update the weights using the current $\mathbf{z}^{(k)}$.
- The layered architecture means we can (1) compute each $\mathbf{z}^{(k)}$ from $\mathbf{z}^{(k-1)}$ and (2) we can use the weight updates computed in layers $K, \dots, k+1$ to update weights in layer k.

Not examinable.