ST449 Artificial Intelligence and Deep Learning

Lecture 2

Introduction to neural networks

Milan Vojnovic

https://github.com/lse-st449/lectures

Topics of this lecture

- Single-layer networks
  - Linear discriminant functions
  - XOR problem
  - Perceptron

- Multi-layer perceptron
  - XOR problem solved

- Feedforward neural networks
  - Basic concepts: architecture, neurons, layers, size, width, depth
  - The expressive power of neural networks
  - A brief history of feedforward neural networks

Single-layer networks

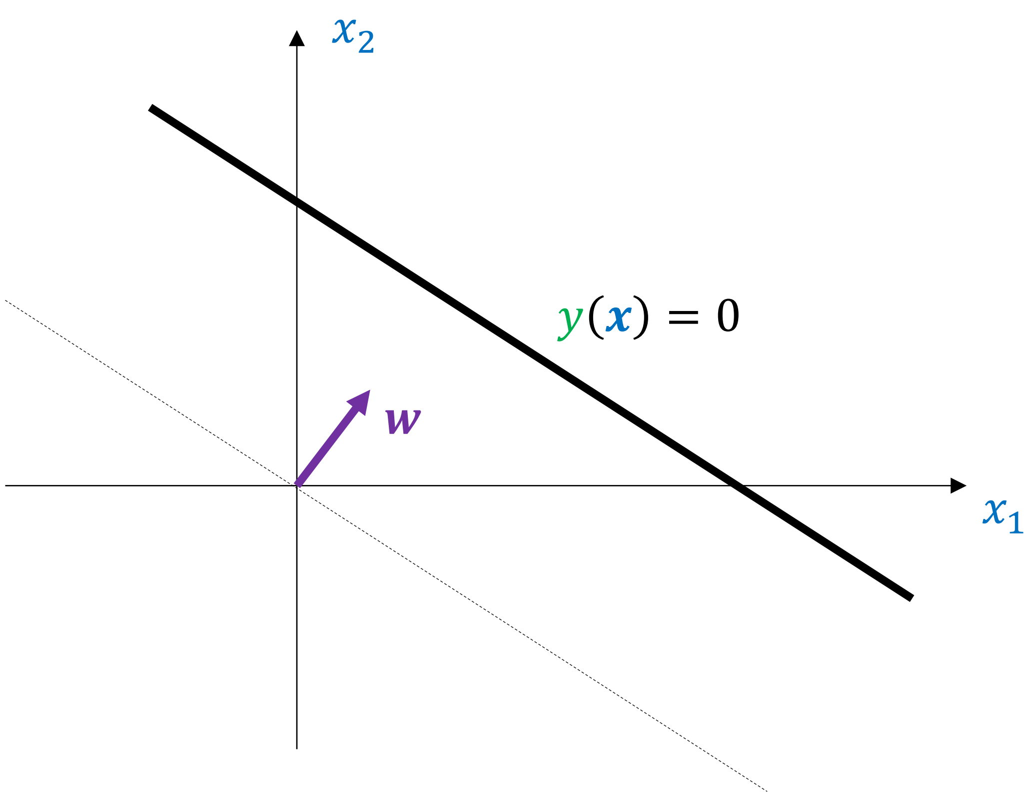
Binary classification problem

- Points $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_m, y_m)$
  - Feature vector: $\boldsymbol{x}_i \in \mathbf{R}^n$
  - Label: $y_i \in \{0,1\}$

- Linear discriminant function:

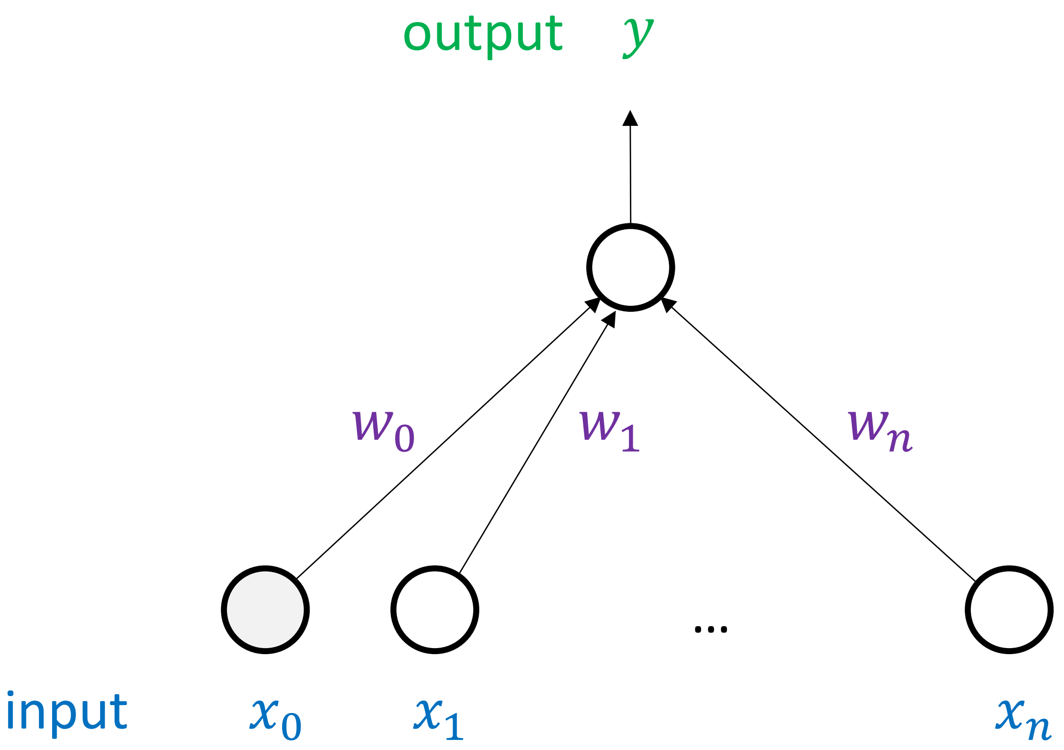
$$y(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w} + w_0$$

weight vector   bias term



- Linear decision boundary
  - Decision boundary $y(\boldsymbol{x}) = 0$ is a $(n-1)$-dimensional hyperplane

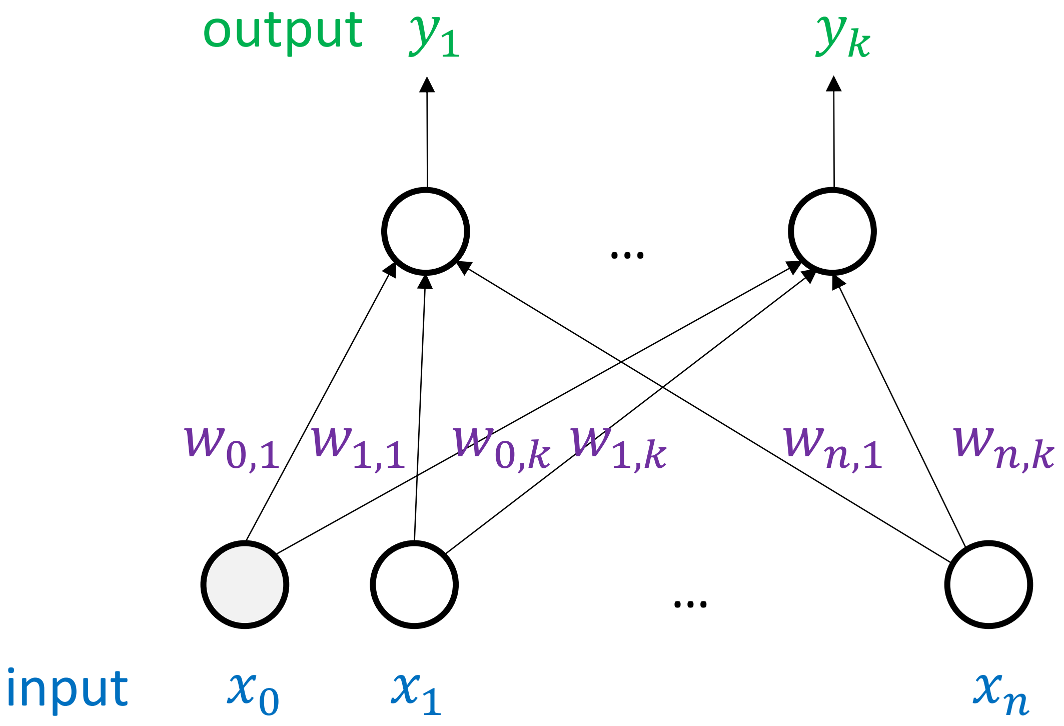
Representation by a neural network diagram



Multiple classes

- If there are $k \geq 2$ classes, for each class we may use a linear discriminant function:

$$y_i(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w}_i + w_{0,i} \quad \text{for } i = 1,2,\dots,k$$



Non-linear discriminant functions

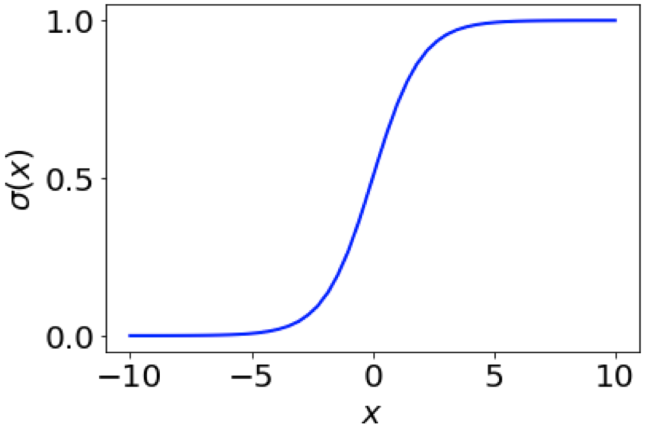
- A way to have a non-linear discriminant function:

$$y(\boldsymbol{x}) = a(\boldsymbol{x}^\top \boldsymbol{w} + w_0)$$

activation function

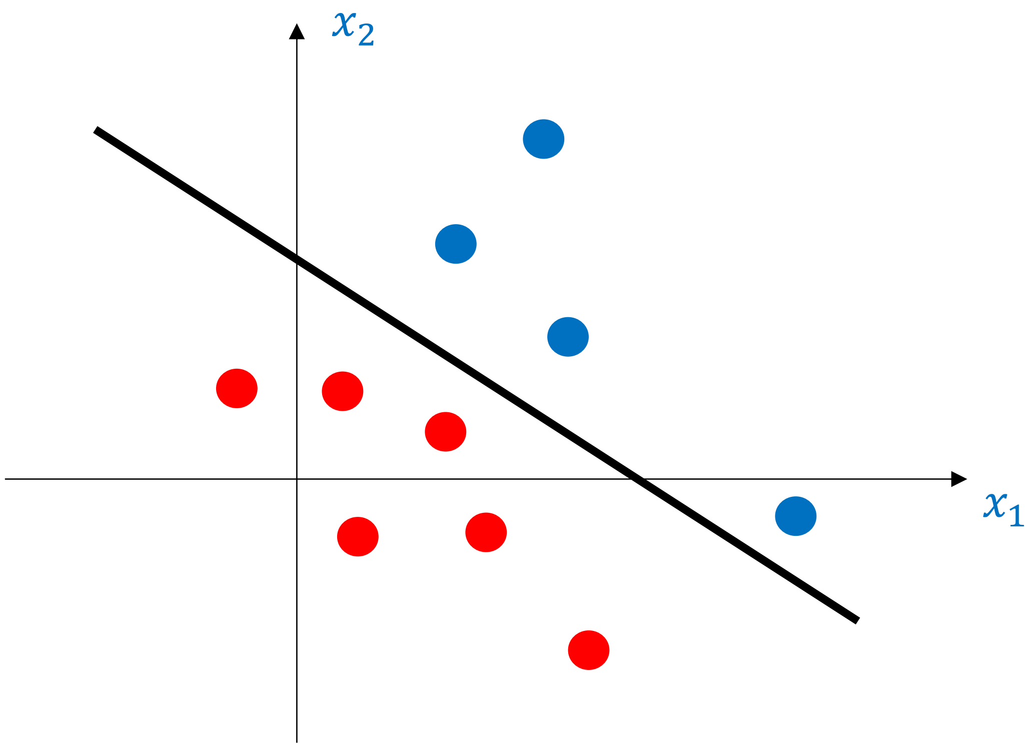
- $a : \mathbf{R} \to \mathbf{R}$ is a monotonic function

- Logistic regression example
  - $a(x)$ is the sigmoid function: $\sigma(x) = 1/(1 + e^{-x})$



Linear separability

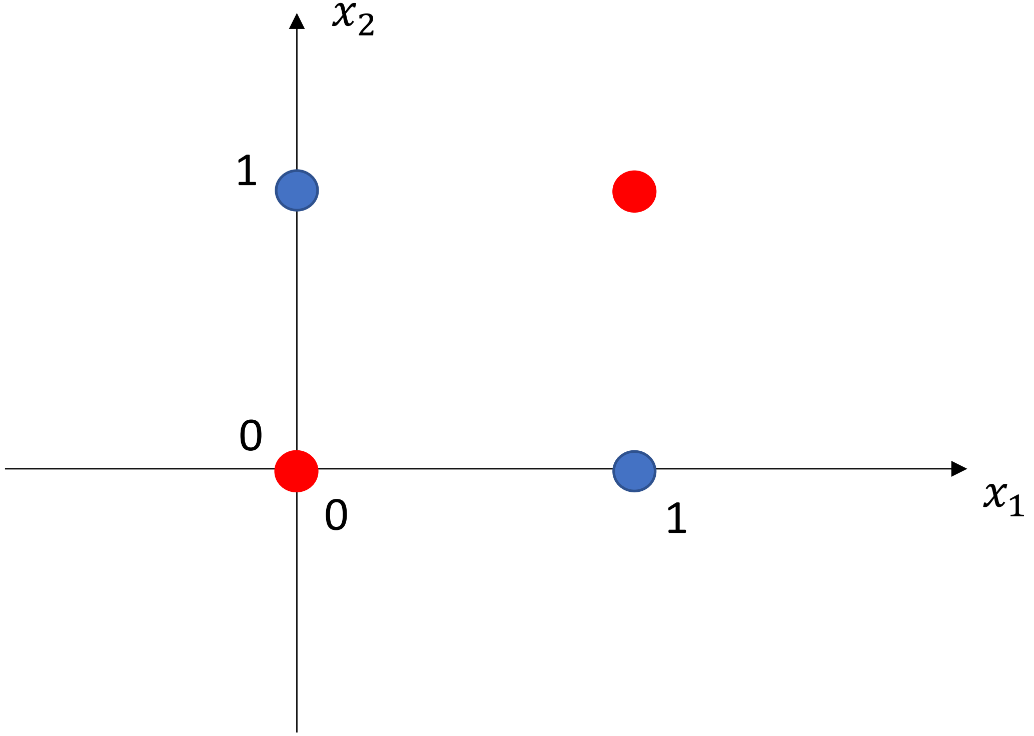
- A set of points is set to be linearly separable if the points belonging to different classes can be separated by a linear decision boundary



- Not all sets of points are linearly separable (e.g. XOR function in the next slide)

XOR: the exclusive-OR

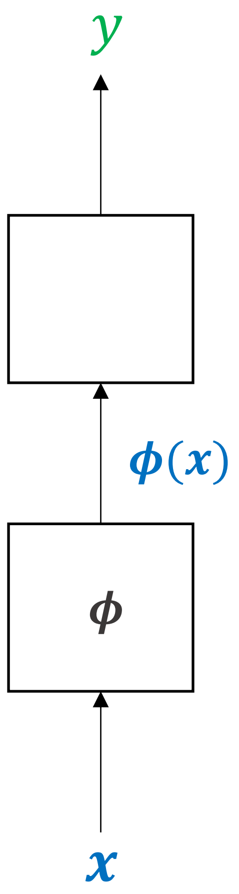
x_1	x_2	$y(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0



- Not linearly separable

Input feature transformation

- Instead of using original feature vectors use transformed feature vectors
  - Representation for a function learning $x \mapsto y$



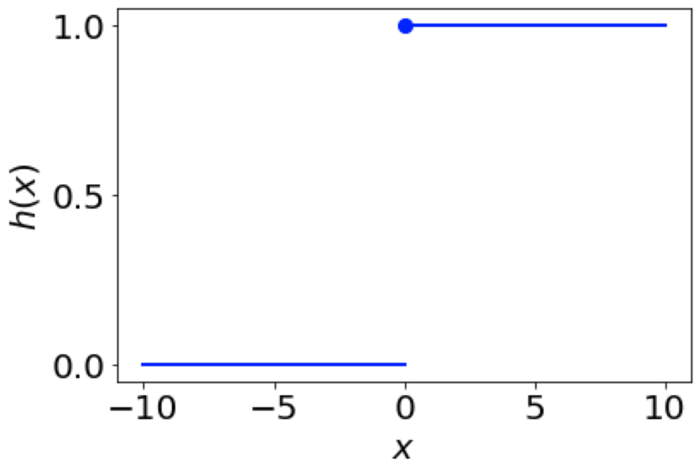
Ex. $\boldsymbol{\phi}$ obtained by a feature engineering

The perceptron

- Perceptron: a single-layer network with a threshold activation function

$$y(\boldsymbol{x}) = h(\textstyle\sum_{i=0}^{n} \phi_i(\boldsymbol{x}) w_i)$$

- Threshold function: $h(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$
  - Also known as a unit step function or the Heaviside step function



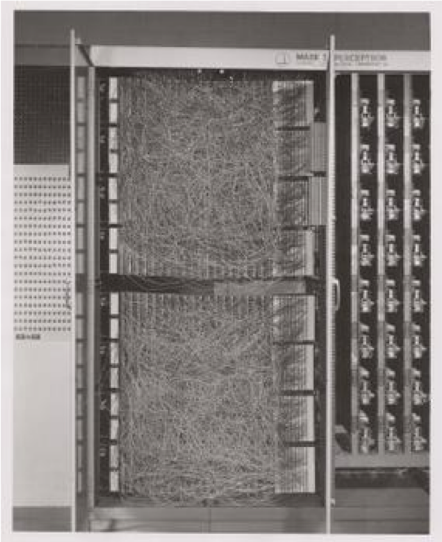
- An alternative half-maximum convention: redefine $h(0) = 1/2$
- In vector notation: $y(\boldsymbol{x}) = h(\boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{w})$

Note: by convention, $\phi_0(\boldsymbol{x}) = 1$

History of perceptron

- The perceptron algorithm was invented by Frank RosenBlatt (psychologist) in 1957 at the Cornell Aeronautical Laboratory

- Intended to be a machine rather than a program
- First implementation in software on an IBM system
- Subsequent implementation in a custom-built hardware: Mark 1 perceptron
  - Patchboard for experimenting with different combinations of features
  - Potentiometers for adjusting weights



Source: Perceptron, Wikipedia

Perceptron criterion

- Convention: $y_i \in \{-1,1\}$ and $h(x) = -1$ if $x < 0$, $h(x) = 1$ otherwise
- Correct classification: $y_i\, \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w} > 0$
- False classification: $y_i\, \underbrace{\boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}}_{\text{same sign as for the predicted label}} < 0$

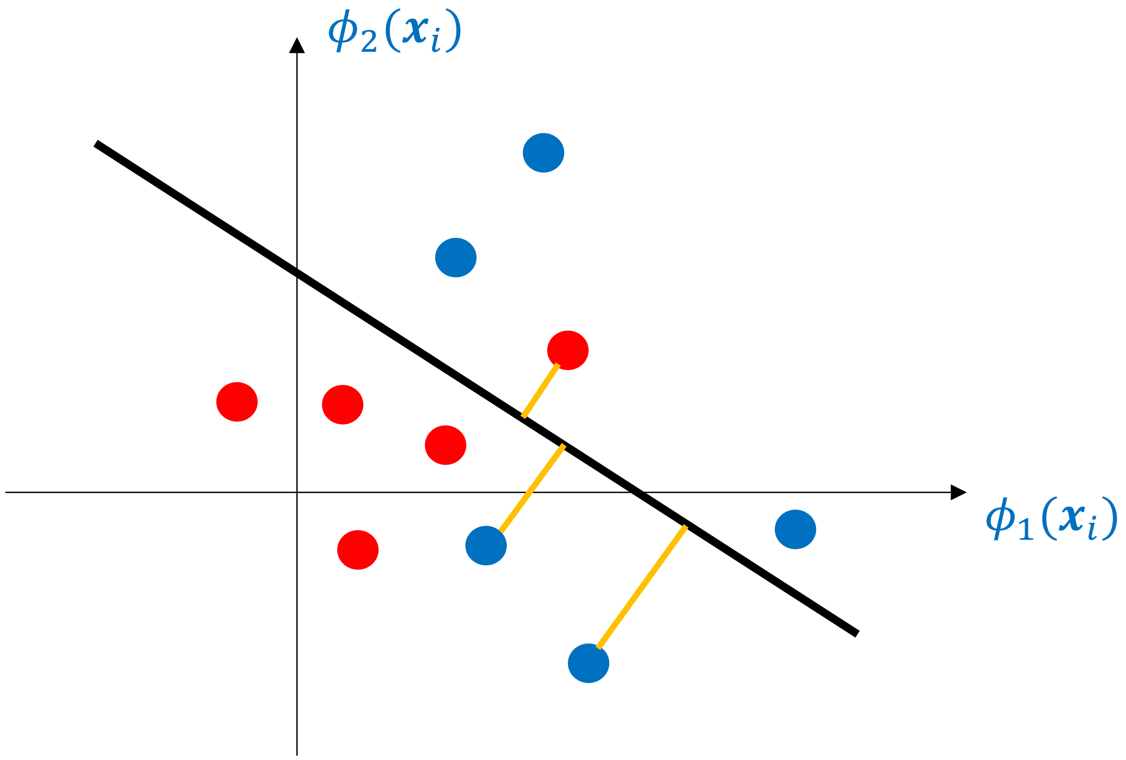
- Goal: find parameter $\boldsymbol{w} \in \mathbf{R}^n$ such that for all input points $y_i\, \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w} > 0$

- Perceptron criterion: loss function

$$L(\boldsymbol{w}) = \sum_{i=1}^{m} \max(-y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}, 0)$$

A continuous piecewise-linear loss function (positive costs for misclassified examples)

Perceptron criterion (cont'd)



- Perceptron criterion loss function is equal to the sum of absolute distances of misclassified examples to the decision boundary

Perceptron learning algorithm

Initialization: $t = 0$, η step size to a positive value, $\boldsymbol{w}^{(0)}$ arbitrarily (say null vector)

Repeat until there are no misclassified examples: // i.e. until $\forall\ (\boldsymbol{x}_i, y_i)$: $y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}^{(t)} > 0$

$\hat{y}_{(i)} = h\left(\boldsymbol{\phi}\left(x_{i(t)}\right)^\top \boldsymbol{w}^{(t)}\right)$ // $(\boldsymbol{x}_{i(t)}, y_{i(t)})$ is the input example at t

If $\hat{y}_{i(t)} \neq y_{i(t)}$ **then**

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \eta y_{i(t)} \boldsymbol{\phi}\left(\boldsymbol{x}_{i(t)}\right)$$

else

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)}$$

end if

$t \leftarrow t + 1$

Perceptron convergence property

- Let $L_i(\boldsymbol{w}) = \max(-y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}, 0)$ be the contribution to the loss function by an input example $(\boldsymbol{x}_i, y_i)$ for given parameter vector $\boldsymbol{w}$

- Convergence property: for every t such that $\hat{y}_{i(t)} \neq y_{i(t)}$:

$$L_{i(t)}\left(\boldsymbol{w}^{(t+1)}\right) < L_{i(t)}\left(\boldsymbol{w}^{(t)}\right)$$

- Exercise: prove this inequality

Solution to the question

- Recall that $L(\boldsymbol{w}) = \sum_{i=1}^{m} L_i(\boldsymbol{w})$ where $L_i(\boldsymbol{w}) = \max(-y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}, 0)$

- For every iteration t such that $\hat{y}_{i(t)} \neq y_{i(t)}$ we have

$$
\begin{aligned}
L_{i(t)}\left(\boldsymbol{w}^{(t+1)}\right) &= \max\left(-y_{i(t)} \boldsymbol{\phi}\left(\boldsymbol{x}_{i(t)}\right)^\top \boldsymbol{w}^{(t+1)}, 0\right) \\
&= \max\left(-y_{i(t)} \boldsymbol{\phi}\left(\boldsymbol{x}_{i(t)}\right)^\top \boldsymbol{w}^{(t)} - \eta \left\|\boldsymbol{\phi}\left(\boldsymbol{x}_{i(t)}\right)\right\|^2, 0\right) \\
&< \max\left(-y_{i(t)} \boldsymbol{\phi}\left(\boldsymbol{x}_{i(t)}\right)^\top \boldsymbol{w}^{(t)}, 0\right) \\
&= L_{i(t)}(\boldsymbol{w}^{(t)})
\end{aligned}
$$

Perceptron convergence theorem

- **Thm** Assume that the algorithm receives a sequence of input examples that are misclassified according to the current parameter vector estimate.

  For any linearly separable set of input examples, the algorithm finds a solution in a finite number of steps.

- Proof is by contradiction (see next slide)

Note: the condition of the theorem can be relaxed by allowing for a bounded number of iterations between any two consecutive misclassified examples input to the algorithm

Proof of the convergence theorem

- Assume that examples are linearly separable:

$$\exists\, \boldsymbol{w}^* :\ y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}^* > 0 \text{ for all } i = 1,2,\dots,m$$

- Note the identity:

$$\boldsymbol{w}^{(t)} = \eta \sum_{i=1}^{m} M_i^{(t)} y_i \boldsymbol{\phi}(\boldsymbol{x}_i)$$

where $M_i^{(t)}$ is the number of iterations s such that $\left(\boldsymbol{x}_{i(s)}, y_{i(s)}\right) = (\boldsymbol{x}_i, y_i)$, $\hat{y}_{i(s)} \neq y_{i(s)}$ and $s \leq t$

- Define $M^{(t)}$ to be the number of iterations s such that $\hat{y}_{i(s)} \neq y_{i(s)}$ and $s \leq t$
  - Indeed, $M^{(t)} = \sum_{i=1}^{m} M_i^{(t)}$

Proof of the convergence theorem (cont'd)

- Fact 1: $(\boldsymbol{w}^*)^\top \boldsymbol{w}^{(t)} = \eta \sum_{i=1}^m M_i^{(t)} y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}^* \geq M^{(t)}\, \eta \min_i y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}^*$
- Fact 2: $\left\|\boldsymbol{w}^{(t)}\right\| \leq \sqrt{M^{(t)}}\, \eta \max_i \|\boldsymbol{\phi}(\boldsymbol{x}_i)\|$
- Fact 3: $(\boldsymbol{w}^*)^\top \boldsymbol{w}^{(t)} \leq \|\boldsymbol{w}^*\| \left\|\boldsymbol{w}^{(t)}\right\|$ (Cauchy-Schwartz)
- Fact 4: $M^{(t)} = t$ (assumption of the theorem)
- From Facts 1 to 4,

$$C_1 t \leq C_2 \sqrt{t} \qquad (\star)$$

  where $C_1 = \min_i y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}^*$ and $C_2 = \|\boldsymbol{w}^*\| \max_i \|\boldsymbol{\phi}(\boldsymbol{x}_i)\|$ are positive constants
- To contradict, assume that the algorithm does not converge in a finite number of iterations: this means that $(\star)$ holds for all $t \geq 0$ which is a contradiction

Note: to check Fact 2 use: $\left\|\boldsymbol{w}^{(t+1)}\right\|^2 = \left\|\boldsymbol{w}^{(t)}\right\|^2 + 2\eta y_{i(t)} \boldsymbol{\phi}\left(\boldsymbol{x}_{i(t)}\right)^\top \boldsymbol{w}^{(t)} + \eta^2 \left\|\boldsymbol{\phi}\left(\boldsymbol{x}_{i(t)}\right)\right\|^2$
and $y_{i(t)} \boldsymbol{\phi}\left(\boldsymbol{x}_{i(t)}\right)^\top \boldsymbol{w}^{(t)} < 0$

Bound on mistakes theorem

- Thm. Assume that training examples are consistent with a separating hyperplane defined by $\boldsymbol{w}^*$, i.e. $y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}^* > 0$ for all $i = 1,2, \dots, m$, and assume that $\|\boldsymbol{\phi}(\boldsymbol{x}_i)\| \leq r$ for all $i = 1,2 \dots, m$

  Then, for the perceptron learning algorithm, we have

$$M^{(t)} \leq \frac{r^2}{\gamma^2}$$

  where $\gamma = \min\left\{\frac{|\boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}^*|}{\|\boldsymbol{w}^*\|} : i = 1,2, \dots, m\right\}$

- Note: if the algorithm receives input examples such that each is misclassified according to the current parameter vector, then $M^{(t)} = t$ and thus in this case the algorithm provides a bound on the number of iterations until convergence

Proof of the theorem

- Similar arguments as for the theorem on slide 18

- Without loss of generality assume that $\|\boldsymbol{w}^*\| = 1$

- From Fact 1 [slide 20]: $(\boldsymbol{w}^*)^T \boldsymbol{w}^{(t)} \geq \gamma\ \eta\ M^{(t)}$
- From Fact 2 [slide 20]: $\left\|\boldsymbol{w}^{(t)}\right\| \leq \eta\ r\sqrt{M^{(t)}}$
- From Fact 3 [slide 20] and $\|\boldsymbol{w}^*\| = 1$: $(\boldsymbol{w}^*)^T \boldsymbol{w}^{(t)} \leq \left\|\boldsymbol{w}^{(t)}\right\|$

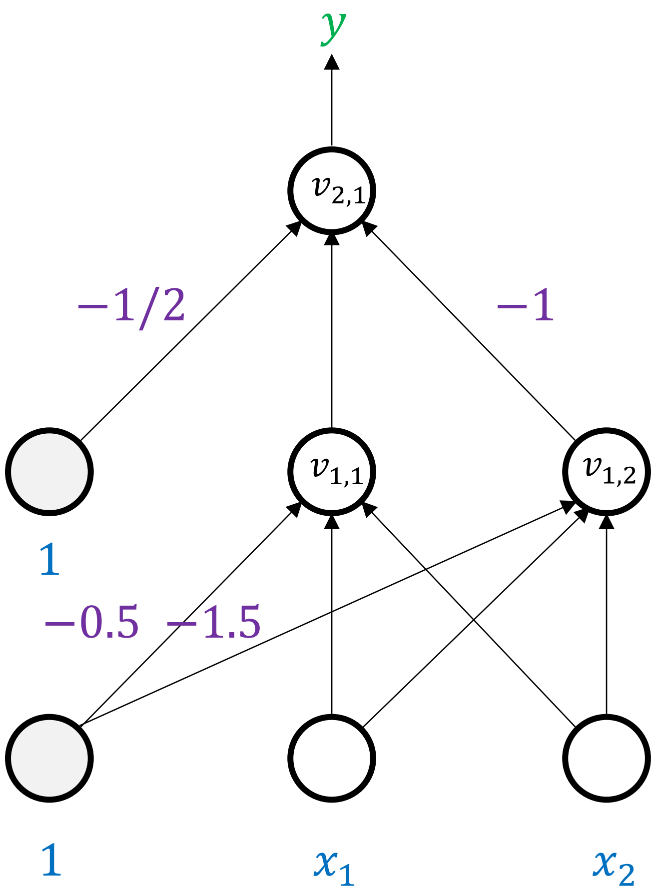
- The claim of the theorem follows from the last three inequalities

Multi-layer perceptron

Multi-layer perceptron

- Single-layer perceptron can only discriminate linearly separable data points
- Extension to non linearly separable data points by adding additional layers
- We next show that a multi-layer perceptron can solve the XOR problem
  - By adding one layer to a single-layer perceptron

The XOR problem solved



extra layer

may think of it as
an implementation of
the transformation $\boldsymbol{\phi}$

- Edge weights are equal to 1 unless otherwise indicated
- Activation function a is the threshold function mapping to $\{0,1\}$, applied by all non-input nodes
- Exercise: check that this two-layer network solves the XOR problem

Exercise solution

- Output of node $v_{1,1}$:

$$a(x_1 + x_2 - 0.5)$$

- Output of node $v_{1,2}$:

$$a(x_1 + x_2 - 1.5)$$

- Output of the network:

$$y(x_1, x_2) = a(1 \times a(x_1 + x_2 - 0.5) - 1 \times a(x_1 + x_2 - 1.5) - 0.5)$$

- Easy to check for each of four possible inputs that y is the XOR function

Feedforward neural networks

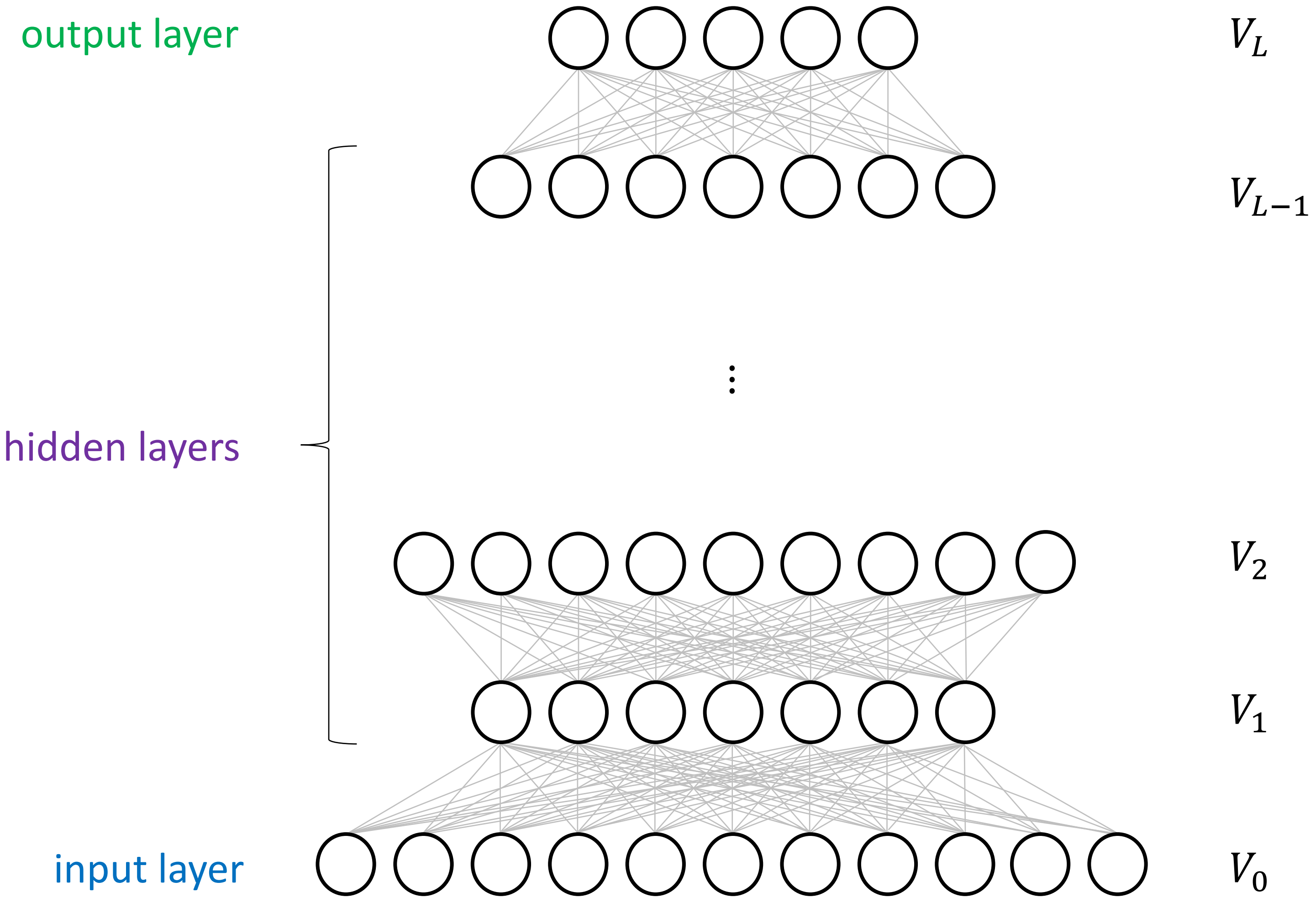
Feedforward networks: basic concepts

- A feedforward network is a network of nodes that can be partitioned in ordered layers such that there may be an edge between two nodes only if they belong to successive layers
- Feedforward network: defined by a direct acyclic graph $G = (V, E)$ and a weight function $w: E \to \mathbf{R}$
- Neuron: represented by a node
- Activation function: each neuron has an activation function $a: \mathbf{R} \to \mathbf{R}$
- Network architecture: defined by the triplet (V, E, a)

Feedforward networks: basic concepts (cont'd)

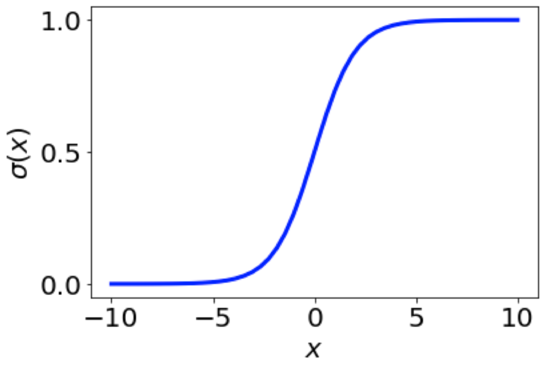
- Network layers defined by a partition of the nodes: $V_0, V_1, \dots, V_L$
  - Input layer: V_0
  - Hidden layers: $V_1, V_2, \dots, V_{L-1}$
  - Output layer: V_L
- Key network properties:
  - Depth: L
  - Size: $|V|$
  - Width: $\max\{|V_0|, |V_1|, \dots, |V_L|\}$
- Comments:
  - Width of the input layer for an n dimensional input space: $|V_0| = n + 1$
  - Each edge $e \in E$ connects some nodes in V_{l-1} and V_l for some layer l

Feedforward networks: basic concepts (cont'd)

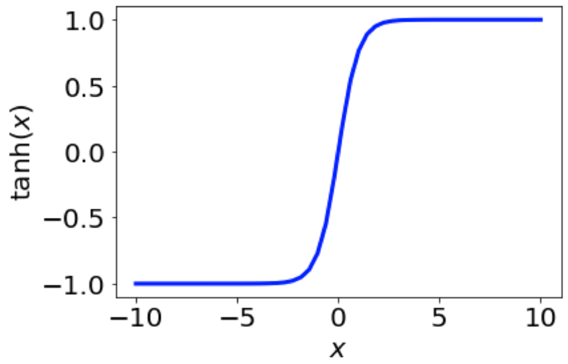


Common activation functions

sigmoid

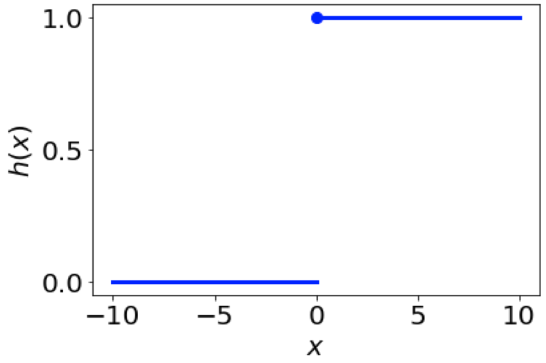


tanh

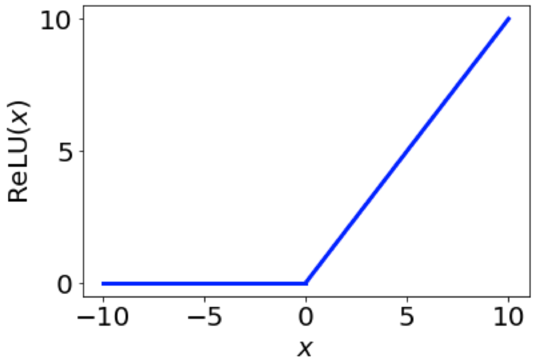


threshold

[unit step,
Heaviside]



rectified linear unit
(ReLU)



Note: $\tanh(x) = 2\sigma(2x) - 1$

Function composition representation

- Function $h(\boldsymbol{x})$ can be expressed as follows:

$$\boldsymbol{y}^{(1)} = a_1(\boldsymbol{W}^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)})$$
$$\boldsymbol{y}^{(2)} = a_2\left(\boldsymbol{W}^{(1)}\boldsymbol{y}^{(1)} + \boldsymbol{b}^{(1)}\right)$$
$$\vdots$$
$$\boldsymbol{y}^{(L-1)} = a_{L-1}\left(\boldsymbol{W}^{(L-2)}\boldsymbol{y}^{(L-2)} + \boldsymbol{b}^{(L-2)}\right)$$
$$\boldsymbol{y} = \boldsymbol{W}^{(L-1)}\boldsymbol{y}^{(L-1)} + \boldsymbol{b}^{(L-1)}$$

- Using the notation $h_l(\boldsymbol{x}) = a_l(\boldsymbol{W}^{(l-1)}\boldsymbol{x} + \boldsymbol{b}^{(l-1)})$, we have

$$h(\boldsymbol{x}) = h_1 \circ h_2 \circ \cdots \circ h_L(\boldsymbol{x})$$

Expressiveness of neural networks

- Function $h_{V,E,a,w}: \mathbf{R}^{|V_0|-1} \to \mathbf{R}^{|V_L|}$

- Hypothesis class of functions:

$$H_{V,E,a} = \{h_{V,E,a,w}: w \text{ is a mapping from } E \text{ to } \mathbf{R}\}$$

- The expressive power of neural networks:
  - What classes of functions can be implemented by using a neural network?
  - For a given network architecture (V, E, a) what functions can be implemented as a function of the network size?

Boolean functions

- Special class of functions: Boolean functions $g\colon \{\pm 1\}^p \to \{\pm 1\}^q$
- The expressiveness power question:
  - What type of Boolean functions can be implemented by $H_{V,E,sign}$?
- For every computer that stores real numbers using b bits, calculating function $f: \mathbf{R}^n \to \mathbf{R}$ corresponds to calculating a Boolean function from $\{\pm 1\}^{nb}$ to $\{\pm 1\}^b$
  - Boolean functions are of practical interest

Boolean functions by neural networks

- Thm. For every n there exists a graph (V, E) of depth 2 such that $H_{V,E,\mathrm{sign}}$ contains all functions from $\{\pm 1\}^n$ to $\{\pm 1\}$

Proof

- Let $f : \{\pm 1\}^n \to \{\pm 1\}$ be a Boolean function
- Let $\boldsymbol{u}_1, \boldsymbol{u}_2, \dots, \boldsymbol{u}_k$ be all vectors in $\{\pm 1\}^n$ for which f outputs 1
- Fact: for every $\boldsymbol{x} \in \{\pm 1\}^n$

$$\boldsymbol{u}_i^\top \boldsymbol{x} \begin{cases} \leq n-2 & \text{if } \boldsymbol{x} \neq \boldsymbol{u}_i \\ = n & \text{if } \boldsymbol{x} = \boldsymbol{u}_i \end{cases}$$

$$\Rightarrow \quad g_i(\boldsymbol{x}) = \operatorname{sign}\left(\boldsymbol{u}_i^T \boldsymbol{x} - n + 1\right) = 1 \quad \Leftrightarrow \quad \boldsymbol{x} = \boldsymbol{u}_i$$

Proof cont'd

- The neural network weight parameters between layers 0 and 1 can be chosen such that each neuron $v_{1,i}$ implements g_i

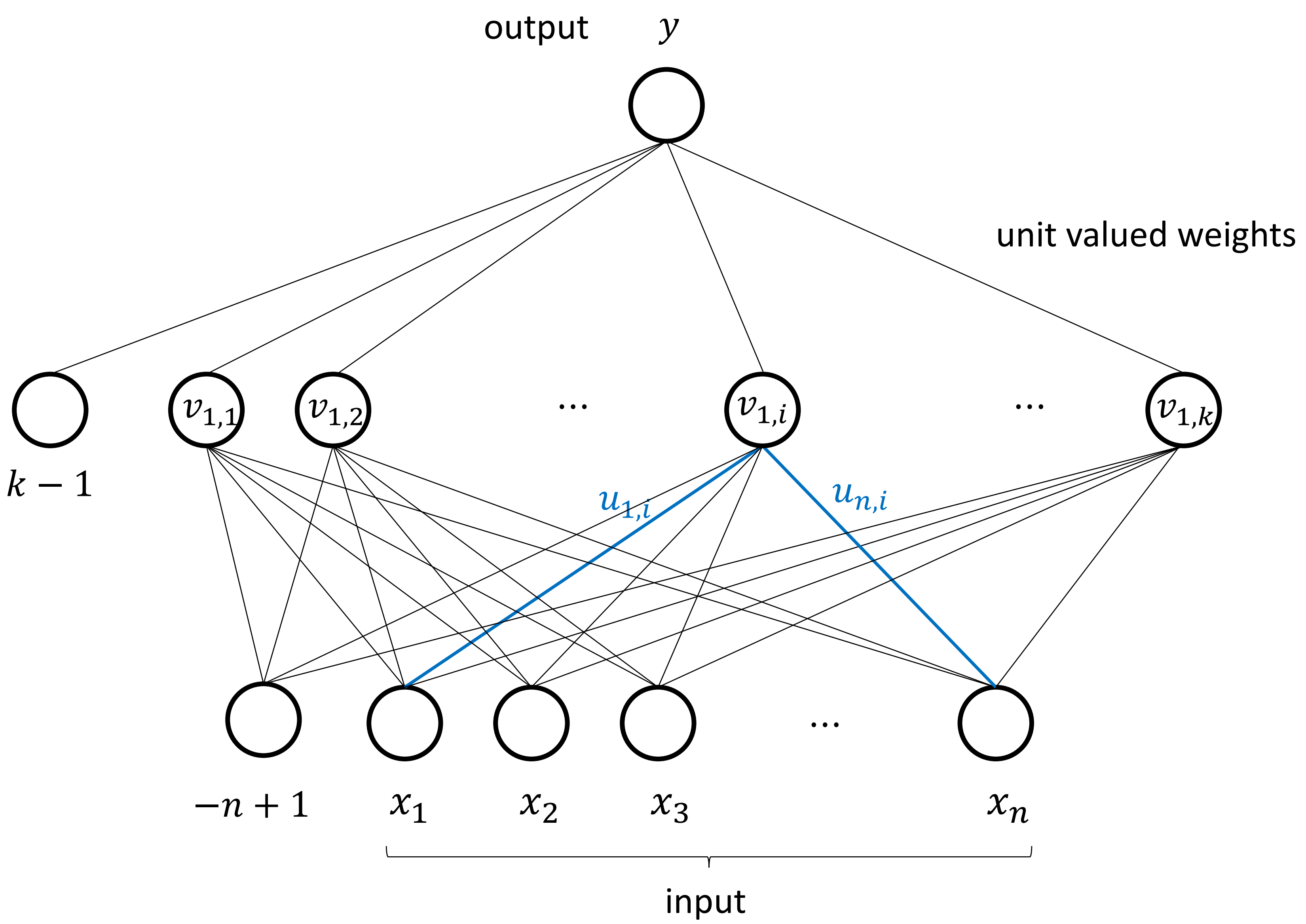
- f is a disjunction of functions $g_1, \dots, g_k$:

$$f(\boldsymbol{x}) = \begin{cases} -1 & if\ g_1(\boldsymbol{x}) = g_2(\boldsymbol{x}) = \cdots = g_k(\boldsymbol{x}) = -1 \\ 1 & \text{otherwise} \end{cases}$$

- Therefore, f can be written as

$$f(\boldsymbol{x}) = \operatorname{sign}\left(\sum_{i=1}^{k} g_i(\boldsymbol{x}) + k - 1\right)$$

The two-layer network used in the proof



Comments

- The last theorem shows that neural networks can implement any Boolean function

- This is a weak property as the network might be exponentially large !

- The proof used a network of depth 2 with

  $|V_0| = n + 1$

  $|V_1| =$ exponential in n if f outputs 1 for a constant fraction of elements in $\{\pm 1\}^n$

  $|V_2| = 1$

Lower bound

- Thm: For every n, let $s(n)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(n)$ such that the hypothesis class $H_{V,E,sign}$ contains all Boolean functions from $\{\pm 1\}^n$ to $\{\pm 1\}$

  Then, $s(n)$ is exponential in n

- Similar result also holds for $H_{V,E,\sigma}$ where σ is the sigmoid function

- Proof uses advanced concepts such as VC dimension

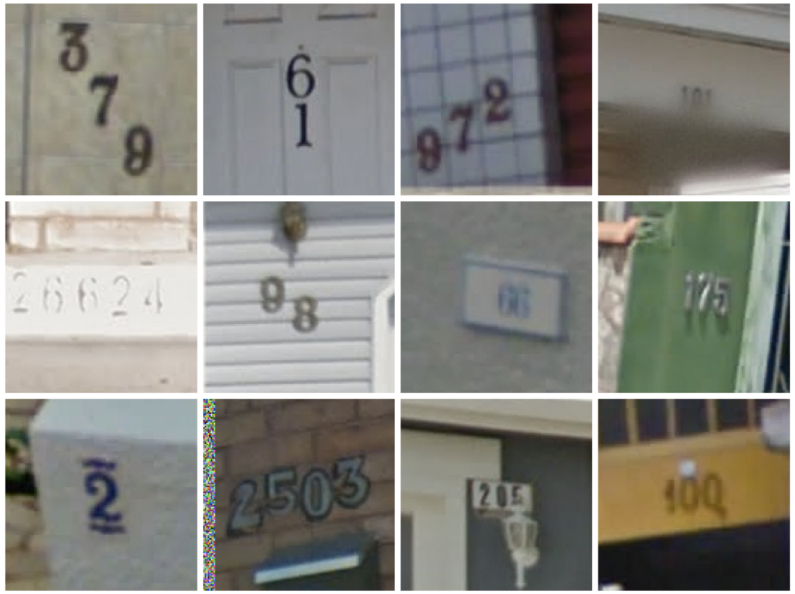
On the expressiveness power of neural networks

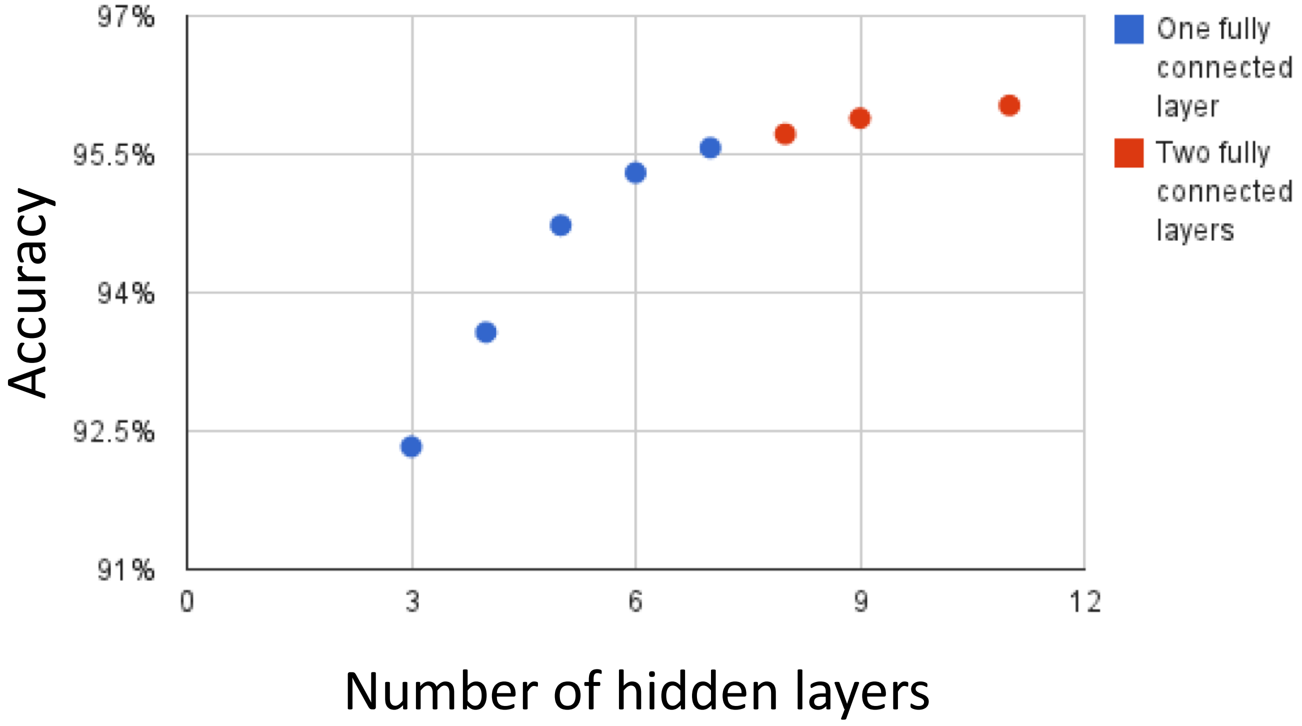
- Universal approximation: sufficiently large depth-2 neural networks, using reasonable activation functions, can approximate any continuous function on a bounded domain [Hornik et al, 1989, …]
  - The required size of such networks can be exponential in the dimension (impractical and prone to overfitting)

- The basic architectural questions for networks of bounded size:
  - How to trade-off between a network width and depth?
  - Should we use networks that are narrow and deep (many layers with a small number of neurons per layer), or shallow and wide?
  - Is the depth really important in deep learning?

- Empirical evidence and intuition: having depth in a neural network is beneficial

Empirical evidence: depth increases accuracy





- Goodfellow et al, Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks, ICLR 2014

Some recent separation results

- There are 3-layer networks of width polynomial in the dimension n which cannot be arbitrarily well approximated by 2-layer networks unless their width is exponential in n
  [Eldan and Shamir, JMLR 2016]

- For any positive integer k, there exist neural networks with $\Theta(k^3)$ layers, $\Theta(1)$ width, and $\Theta(1)$ distinct parameters which cannot be approximated by networks with $\Theta(k)$ layers unless they are exponentially large of size $\Omega(2^k)$
  [Telgarsky, COLT 2016]

- For any positive integer k, there exists a function representable by a ReLU neural network with k^2 hidden layers and total size k^3 such that any ReLU neural network with at most k hidden layers requires at least $\Omega(k^{k+1})$ total size
  [Arora et al, ICLR 2018]

For big O notation, see Big O notation

Historical remarks

- 1940s: function approximation used to motivate ML models such as perceptron
  - Early models based on linear models
  - Minsky’s critique: inability of linear models to learn the XOR function
- The need to learn nonlinear functions led to
  - Multi-layer perceptron
  - Backpropagation: algorithm for computing a loss function gradient [LeCun (1985), Parker (1985), Rumelhart et al (1986)]
- 1980s: feedforward networks gained in popularity
  - Established the core ideas behind modern feedforward neural networks

* Backpropagation algorithm: next week

Historical remarks (cont’d)

- Most improvements in neural network performance attributed to:
  - Large datasets: allowing for generalization
  - Computing power and software: allowing to train larger neural networks

- Algorithmic developments also improved neural network performance:
  - Choice of the loss function: replacement of mean squared error with cross-entropy loss functions
  - Choice of activation functions: replacement of sigmoid activation functions with piecewise linear activation functions (ex. ReLUs)

Exercise: the choice of the loss function

- Consider a binary classifier using a single neural unit with activation function a:

$$\mathbf{Pr}[y_i = 1] = 1 - \mathbf{Pr}[y_i = \mathbf{0}] = p_{\boldsymbol{\theta}} = a(\boldsymbol{x}_i^\top \boldsymbol{w} + b)$$

where $\boldsymbol{\theta}^\top = (\boldsymbol{w}^\top, b)$ is the parameter vector

- Loss functions:
  - Mean squared error: $f_{\mathrm{MSE}}(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - p_{\boldsymbol{\theta}})^2$
  - Cross-entropy: $f_{\mathrm{CE}}(\boldsymbol{\theta}) = -\sum_{i=1}^{m} y_i \log(p_{\boldsymbol{\theta}}) + (1 - y_i) \log(1 - p_{\boldsymbol{\theta}})$

- Compute the gradient vectors for the mean squared error loss function and the cross-entropy loss function (assume a is sigmoid function, for simplicity!)
  - Why the gradient of the cross-entropy loss function may be preferred?

Exercise solution

- For the mean squared error loss function:

$$\frac{\partial}{\partial w_j} f_{\mathrm{MSE}}(\boldsymbol{w}, b) = -\sum_{i=1}^{m} \underbrace{a'\left(\boldsymbol{x}_i^{\top}\boldsymbol{w} + b\right)}_{\text{diminishing to zero for } |\boldsymbol{x}_i^{\top}\boldsymbol{w} + b| \text{ large}} \left(y_i - a\left(\boldsymbol{x}_i^{\top}\boldsymbol{w} + b\right)\right) x_{i,j}$$

- For the cross-entropy loss function:

$$\frac{\partial}{\partial w_j} f_{\mathrm{CE}}(\boldsymbol{w}, b) = -\sum_{i=1}^{m} \left(y_i - a\left(\boldsymbol{x}_i^{\top}\boldsymbol{w} + b\right)\right) x_{i,j}$$

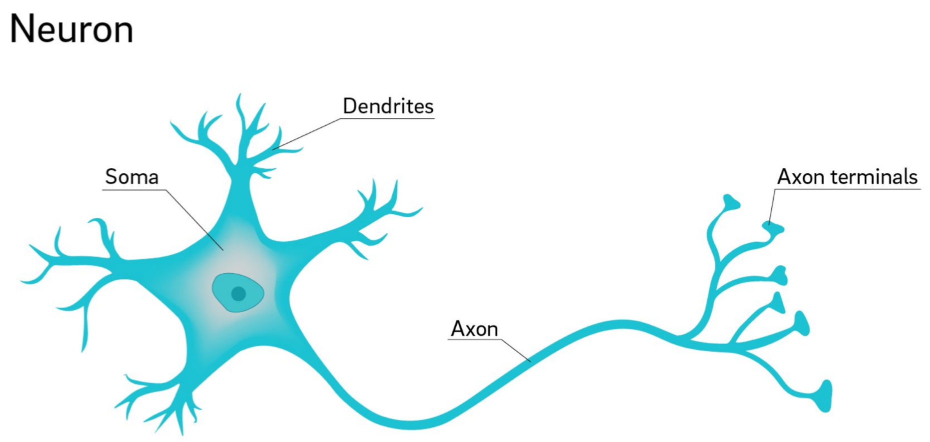
Use of ReLU activation functions

- ReLUs used in early neural networks
  - Cognitron and neocognitron [Fukushima, 1975, 1980]
- Largely replaced by sigmoid function in 1980s
  - Perhaps sigmoids perform better for small neural networks
- Avoided until early 2000s
  - Preference for differentiable functions
- [Jarrett et al, 2009] found ReLUs to perform better than some other commonly used activation functions
- [Glorot et al, 2011] deep neural network easier with ReLU than with using activation with saturation

Biological justification of ReLUs

- Properties of biological neurons:
  - For some inputs biological neurons are completely inactive
  - For some inputs, the outputs of biological neurons are proportional to their inputs
  - Most of the time biological neurons operate in a region in which they are inactive (sparse activation)

- These properties are captured by ReLUs



References

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville, Deep learning, The MIT Press, 2016
- Christopher M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, 1995
- Shai Shalev-Shwartz and Shai Ben-David, Understanding Machine Learning: from theory to algorithms, Cambridge University Press, 2014

References: expressiveness of neural networks

- Raman Arora et al, Understanding Deep Neural Networks with Rectified Linear Units, ICLR 2018
- Matus Telgarsky, Benefits of depth in neural networks, Journal of Machine Learning Research, Vol 49: 1-23, 2016
- Ronen Eldan and Ohad Shamir, The power of depth for feedforward neural networks, Journal of Machine Learning Research, vol 49:1-34, 2016
- Kurz Hornik, Maxwell Stinchcombe and Halber White, Multilayer feedforward networks are universal approximators, Neural Networks, Vol 2, 359-366, 1989

* Theoretical papers

Seminar 2 preview

- Solving the XOR problem in TensorFlow
- Learning new TensorFlow concepts such as scope
- HW: implementation and evaluation of the perceptron learning algorithm

Next lecture

- Training neural networks
  - Gradient descent algorithms
  - Stochastic gradient descent algorithms
  - Backpropagation algorithm
  - Acceleration by momentum
  - Adaptive moment estimation
  - Dropout
  - Batch normalization