know where to look for

- http://neuralnetworksanddeeplearning.com/ Chapter 1
- d2l.ai Chapter 3 and Chapter 4
- https://numpy.org/doc/

What you will see today

- Artificial Neurons
- Neural networks
- theory: understand what that linear model is computing
- how neural networks can represent decision boundaries in classification

Outline

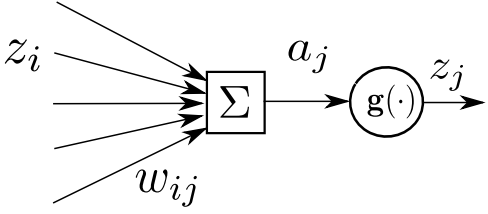
A mathematical function that for instance can:

- Take some inputs $\{z_i\}_{i=1}^{d}$.
- Depend on some weights w_{ij} and bias b_j.
- Apply some non-linear activation function $g(\cdot)$.
- Output z_j.

Forward equation:

$$a_j = \sum_{i=1}^{d} z_i w_{ij} + b_j \qquad \text{linear mapping}$$

$$z_j = g(a_j) \qquad \text{nonlinear activation}$$



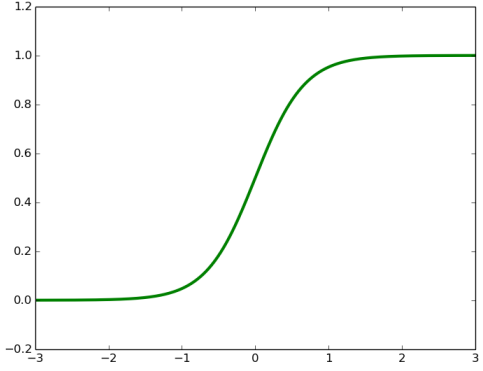
Activation functions

- A function $g : \mathbb{R}^1 \longrightarrow \mathbb{R}^1$
- Introduces non-linearity.
- Should be differentiable if you are planning to use gradient-based optimisation.
- Typically prevents or reduces weak signals from passing through.
- Intuition: Determines what the neuron fires for different outputs of the linear mapping.

Sigmoid activation

We could for instance use the logistic sigmoid function:

$$g(x) = \frac{1}{\exp(-x) + 1}$$



- Called sigmoid activation.
- Historically popular.
- Currently rarely used.

Suffer from gradient saturation: its derivative gets exponentially small for large $|x|$:

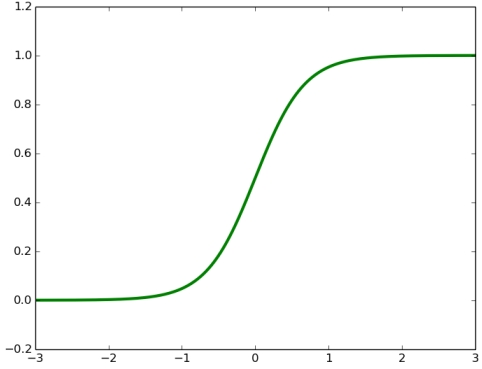
$$\frac{\partial s(x)}{\partial x} = \frac{e^{-x}}{(e^{-x}+1)^2} \leq \begin{cases} e^{-x} & x \geq 0 \\ \frac{1}{e^{-x}+1} & x < 0 \end{cases}$$

Sigmoid activation

We could for instance use the logistic sigmoid function:

$$g(x) = \frac{1}{\exp(-x) + 1}$$

- Called sigmoid activation.
- Historically popular.
- Currently rarely used.



Suffer from gradient saturation: its derivative gets exponentially small for large $|x|$:

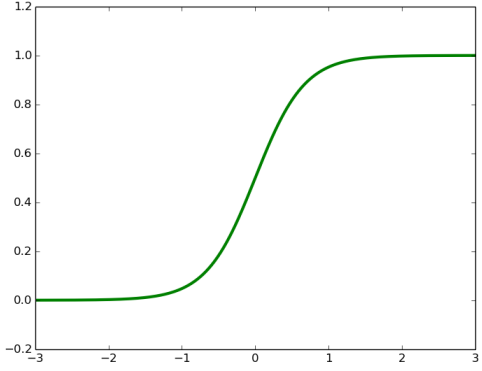
$$\frac{\partial s(x)}{\partial x} = \frac{e^{-x}}{(e^{-x}+1)^2} \leq \begin{cases} e^{-x} & x \geq 0 \\ \frac{1}{e^{-x}+1} & x < 0 \end{cases}$$

Sigmoid activation

We could for instance use the logistic sigmoid function:

$$g(x) = \frac{1}{\exp(-x) + 1}$$

- Called sigmoid activation.
- Historically popular.
- Currently rarely used.



Suffer from gradient saturation: its derivative gets exponentially small for large $|x|$:

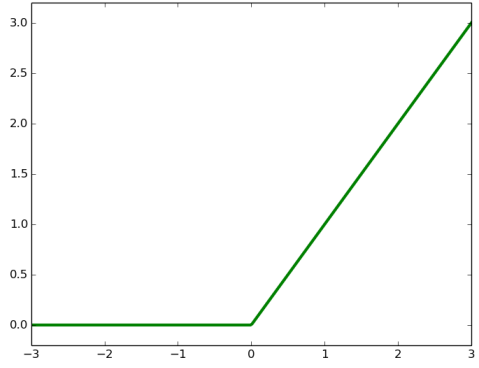
$$\frac{\partial s(x)}{\partial x} = \frac{e^{-x}}{(e^{-x}+1)^2} \leq \begin{cases} e^{-x} & x \geq 0 \\ \frac{1}{e^{-x}+1} & x < 0 \end{cases}$$

Rectified linear unit (ReLU) activation

An alternative is to simply set negative values to 0, i.e. let:

$$g(x) = \max(0, x)$$

- Called ReLU activation.
- Gradient do not saturate.
- Not differentiable in 0.
  - Not of practical concern.
  - Could define it to be 1.

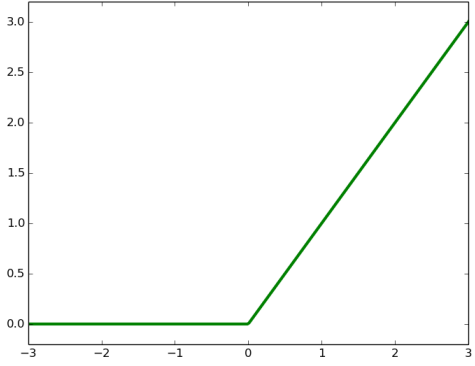


Rectified linear unit (ReLU) activation

The gradient is 0 for negative inputs.

- Might cause inactive nodes to remain inactive.
- Might still work fine in practice.
- There exists alternatives with non-zero gradient for negative inputs, e.g. leaky ReLU: $g(x) = \max(0.01x, x)$

A popular choice, particularly with convolutional neural networks.



ReLU, not leaky one

Artificial (non-spiking) neurons as used in Deep Learning are **not** a model of biological neurons. not even close!.

- Biological neurons are spiking neurons with time-dependency.
- Fire spikes, if their voltage (= difference in energy compared to the outside) has increased over a threshold. The voltage can be increased by currents from the outside. Current = flow rate of charged particles (Na^+, K^+ ions in biological systems, electrons in metals).
- Biological models often model the voltage over time as a function of incoming currents (and voltage decay, a neuron is not a perfect storage for energy)
- the formulas are not for exams:

$$\text{LiF:} = V_{t+\Delta t} - V_t = -\frac{V_t - E_L}{\tau_m}\Delta t + \frac{I_t}{\tau_m g_L}\Delta t$$

$$\text{if } V_{t+\Delta t} > V_0 : \text{ emit spike and reset } V_{t+2\Delta t} := E_L$$

Δt - small time amount, V - Voltage, I - incoming current, E_L - resting potential, τ membrane time constant (how fast V_t decays), g_L Leak conductance

For examples see: https://compneuro.neuromatch.io/tutorials/W2D3_BiologicalNeuronModels/student/W2D3_Tutorial1.html

Compare these teo

Compare above formulation against artificial neurons

- spiking NNs: voltage over time steps. Current spikes.
- ANNs:
  - one-step function computation (but layers as weak analogy to time steps)
  - Affine mapping + nonlinearity as basic building block
  - no physical measures.

$$a_j = g(\boldsymbol{w} \cdot \boldsymbol{z} + b)$$

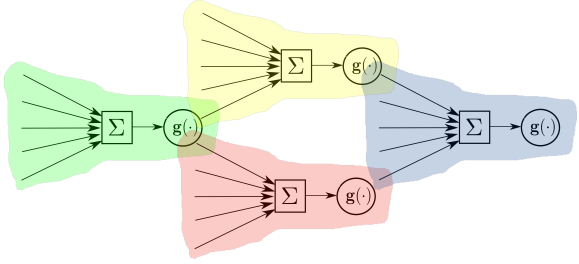
Spiking neural network architectures might be the next generation of computing hardware (more likely than the quantum compute hype) because systems using them might use dramatically less energy.

Energy consumption **is** a deployment cost issue for deep learning models:
https://www.wionews.com/business-economy/chatgpt-costs-700000-daily-creator-openai-may-go-bankrupt-by-2024-report-625306
https://www.firstpost.com/tech/news-analysis/openai-may-go-bankrupt-by-2024-chatgpt-costs-company-700000-dollars-every-day-12986012.html.

For a spiking neuron toolbox:
https://snntorch.readthedocs.io/en/latest/tutorials/tutorial_1.html

A neural network is a directed graph structure made from connected neurons.

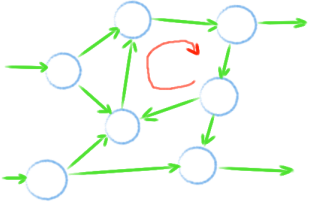
- A neuron can be input to many other neurons.
- A neuron can receive input to many other neurons.
- Each neuron has same structure $(g(\cdot), \sum)$ but different parameters (w_{ij}, b_j).
- Can stack neurons in layers.



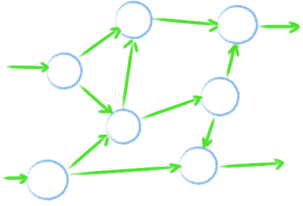
Definition: Neural Network

Any directed graph built from neurons is a neural network.

Two important types: recurrent and feedforward neural networks



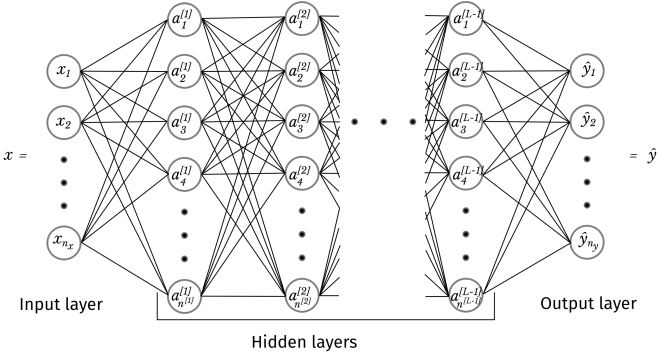
recurrent (not covered in this lecture)
for sequence processing

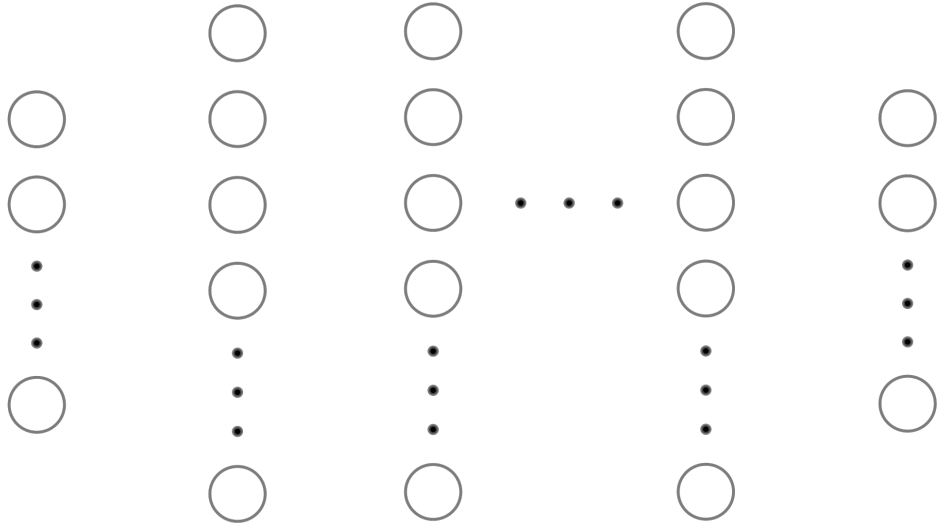


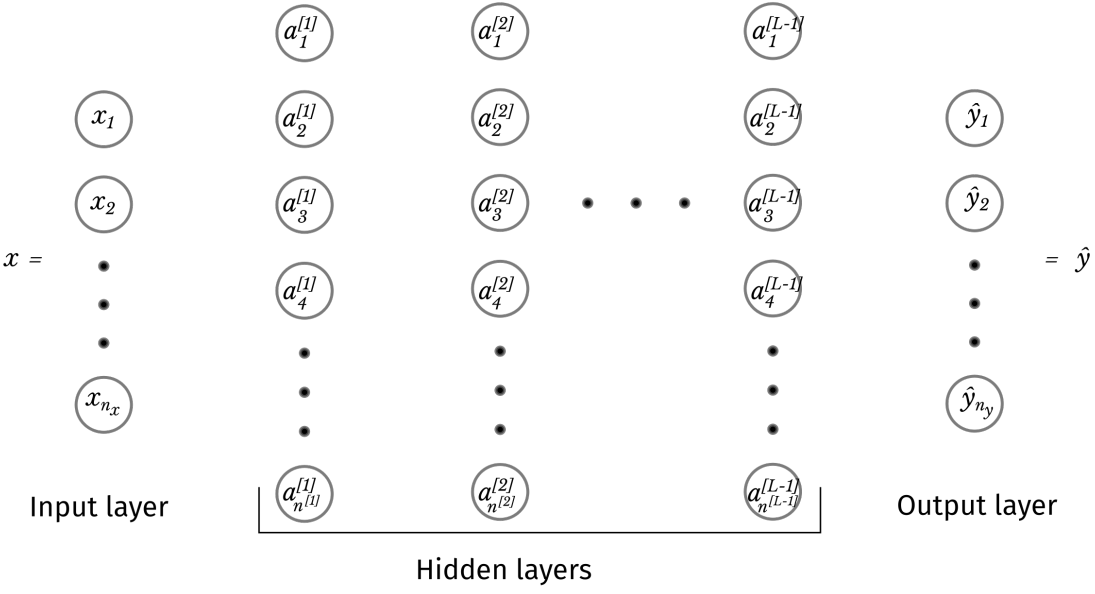
feedforward
e.g. for image classification

layer instead of general graph

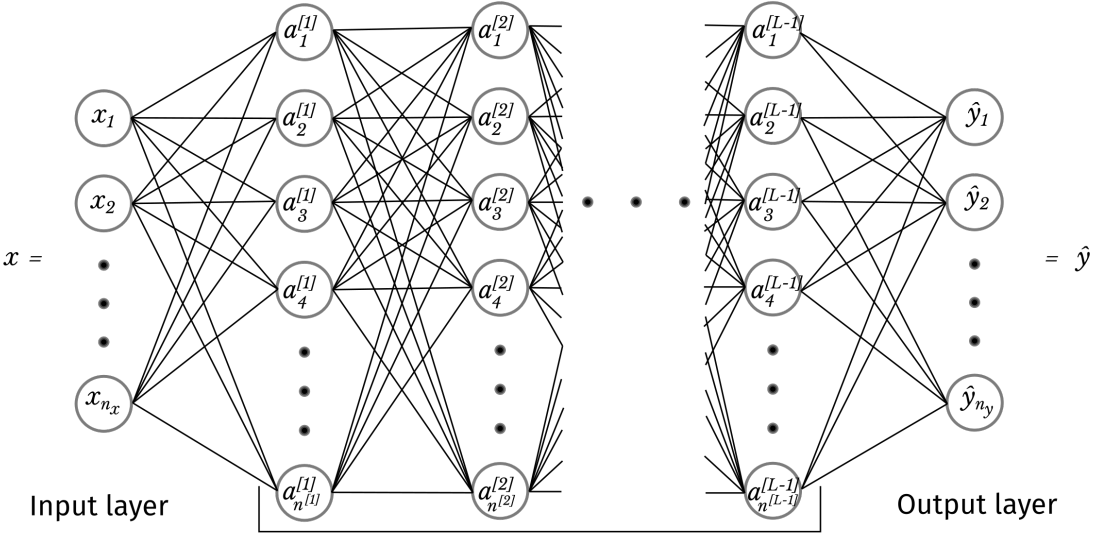
For computation efficiency, one usually organizes neurons in layers .
For the simplest type of networks: one layer is fully connected to the next layer.

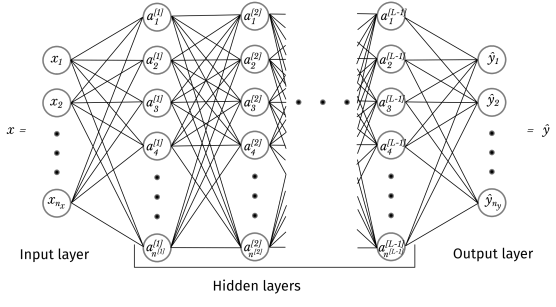





$x = $
x_1
x_2
x_{n_x}
$a_1^{[1]}$ $a_2^{[1]}$ $a_3^{[1]}$ $a_4^{[1]}$ $a_{n^{[1]}}^{[1]}$
$a_1^{[2]}$ $a_2^{[2]}$ $a_3^{[2]}$ $a_4^{[2]}$ $a_{n^{[2]}}^{[2]}$
$a_1^{[L-1]}$ $a_2^{[L-1]}$ $a_3^{[L-1]}$ $a_4^{[L-1]}$ $a_{n^{[L-1]}}^{[L-1]}$
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_{n_y}$
$= \hat{y}$
Input layer
Hidden layers
Output layer

Dense (fully-connected) feedforward neural network



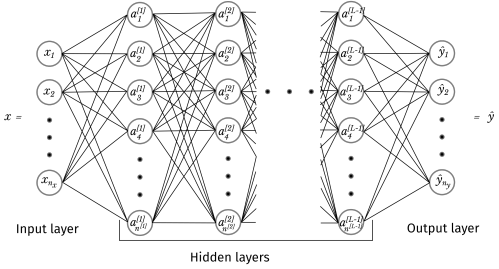


Each neuron (circle) computes this:

$$\text{output} = \text{activation}(\text{inner product}(\boldsymbol{w}, \textbf{input}) + \text{bias})$$

$$a_k^{(l)} = g\left(\boldsymbol{w}_{[:,k]}^{(l)} \cdot \boldsymbol{a}^{(l-1)} + b_k^{(l)}\right),$$

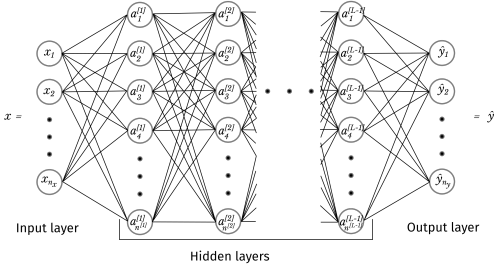
$$(k \in \{1, 2, \ldots, n^{[l]}\}, l \in \{1, 2, \ldots, L\})$$



$$a_k^{(l)} = g\left(\boldsymbol{w}_{[:,k]}^{(l)} \cdot \boldsymbol{a}^{(l-1)} + b_k^{(l)}\right)$$

output $a_k^{(l)}$ for a single neuron is $g(z_k^{(l)})$ where $z_k^{(l)}$ is the result from an inner product between

- the activation vector $\boldsymbol{a}^{(l-1)}$ of the previous layer $l-1$
- the k-th slice $\boldsymbol{w}_{[:,k]}^{(l)}$ of the weight matrix $\boldsymbol{w}^{(l)}$ of the current layer l
- added a bias term $b_k^{(l)}$ to the inner product



$$a_k^{(l)} = g\left(\boldsymbol{w}_{[:,k]}^{(l)} \cdot \boldsymbol{a}^{(l-1)} + b_k^{(l)}\right)$$

- if one concatenates all outputs $a_k^{(l)}$ of layer (l) , and applies $g(\cdot)$ element-wise, then:

$$\boldsymbol{a}^{(l)} = g\left(\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$$

- where $\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)}$ is a matrix-vector multiplication

$$\boldsymbol{a}^{(l)} = g\left(\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$$

- The $\boldsymbol{W}$ and $\boldsymbol{b}$ are "trainable", and will be adjusted according to some optimization routine.
- By convention:
  - $a_k^{(0)} = x_k$ and $a_k^{(L)} = \hat{y}_k$
  - The network has L layers (we do not include the input layer in the count) and $L-1$ hidden layers.

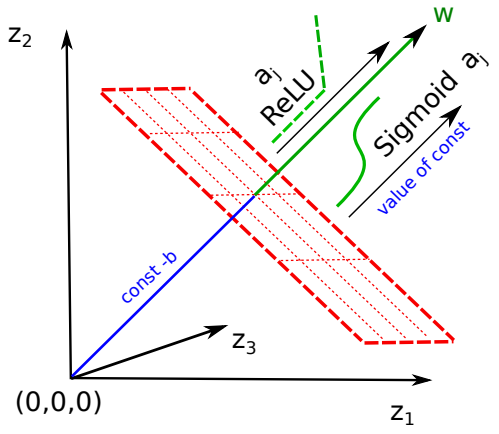
$$\boldsymbol{a}^{(l)} = g\left(\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$$

- The $\boldsymbol{W}$ and $\boldsymbol{b}$ are "trainable", and will be adjusted according to some optimization routine.
- By convention:
  - $a_k^{(0)} = x_k$ and $a_k^{(L)} = \hat{y}_k$
  - The network has L layers (we do not include the input layer in the count) and $L-1$ hidden layers.

- What does a neuron compute?
- What could a neural network compute?

$$f(x) = w \cdot x + b$$

Next: What does the linear model actually represent / compute, once we have a (w, b) ?



- the red plane are the set of points where $f(x) = const$
- The weights w defines the orientation of the plane, b shifts position of it.
- Output of activation function is constant in the red plane.

Goal: understand what the mapping $f(\cdot)$ does.

Approach: characterizing the set of points x with a constant output $f(x) = c$.

Thinking task

What is the set of points $x = (x_1, x_2) \in \mathbb{R}^2$ such that

$$3x_1 - 2x_2 + 3 = 0 \ ?$$

What is the set of points $x = (x_1, x_2, x_3) \in \mathbb{R}^3$ such that

$$x_1 - x_2 - 2x_3 + 2 = 0 \ ?$$

A. What is the set of points x: $f_{w,b}(x) = 0$?
B. What is the set of points x the prediction is a constant c, that is $f_{w,b}(x) = c$?

What is the set of points x: $f_{w,b}(x) = 0$?

$$f_{w,b}(x) = 0 \Leftrightarrow x \cdot w = -b$$

To understand how the bias b influences the zero set, lets consider three cases:

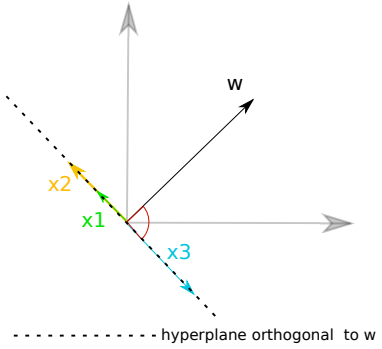
- $b = 0$
- $b > 0$
- $b < 0$

The set of points x: $f_{w,b}(x) = 0$ – The case $b = 0$

- We know that for $b = 0$: $f_{w,0}(x) = w \cdot x = 0$ holds for the zero vector $x = 0$.
- The set of points x such that the inner product

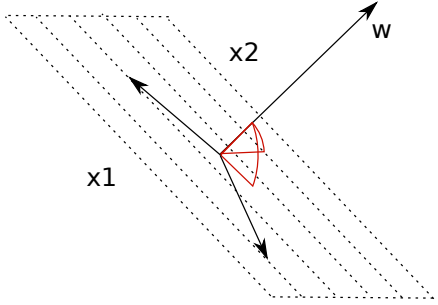
$$x \cdot w = 0$$

is in 2 dims a one-dimensional line, which goes through the origin $(x_1, x_2) = (0, 0)$, and which is orthogonal to w.



x_1,x_2,x_3 are all orthogonal to the vector w

The analogy also holds for 3 or more dimensions. So for 3 dims it is a two-dimensional plane, which goes through the origin $(x_1, x_2, x_3) = (0, 0, 0)$.



For n dimensions the plane of orthogonal vectors has $n - 1$ dimensions (+ goes through the origin), but is still a hyperplane

Recap hyperplane of dimension $n-1$

- P is a hyperplane (affine space) if it holds:
  $x_1 \in P, x_2 \in P, o \in P \Rightarrow a_1(x_1 - o) + a_2(x_2 - o) + o \in P$ (space closed under linear operations)
- can find $n-1$ basis vectors such that each point of P can be represented as an offset plus a linear combination of the basis vectors

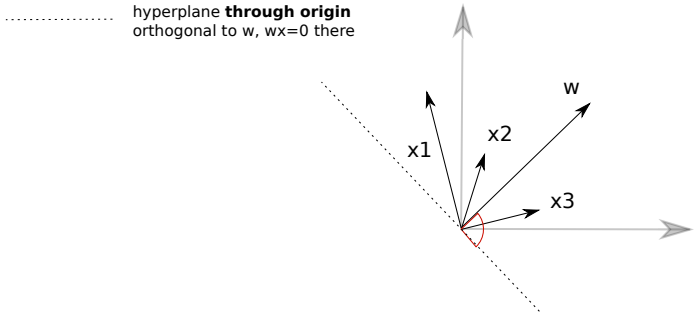
$$\exists\, o, v_1, \ldots, v_{n-1} \text{ such that}$$

$$\forall x \in P\ \exists\, a_1, \ldots, a_{n-1} \text{ such that } x = o + \sum_{i=1}^{n-1} a_i v_i = \boldsymbol{a} \cdot \boldsymbol{V}$$

- o is any vector which lies on the hyperplane
- If the affine hyperplane is a vector space, then it contains the zero-vektor, and one can choose $o = 0$, then the definition simplifies to $x_1 \in P, x_2 \in P \Rightarrow a_1 x_1 + a_2 x_2 \in P$

$$b < 0,\ w \cdot x + b = 0 \Rightarrow w \cdot x = -b > 0$$

We know: $w \cdot x > 0$ for all points x that are on that side of the **hyperplane through the origin orthogonal to** w, in which w points to.

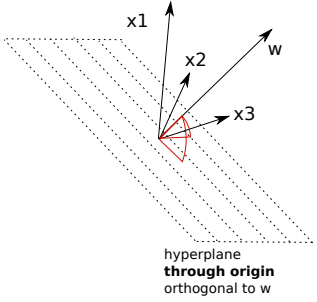


In the above figure $\mathbf{x_1}$,$\mathbf{x_2}$,$\mathbf{x_3}$ all have $w \cdot x_i > 0$ because relative to the hyperplane orthogonal to w which goes through the origin, they all point towards the direction of w

$$b < 0,\ w \cdot x + b = 0 \Rightarrow w \cdot x = -b > 0$$

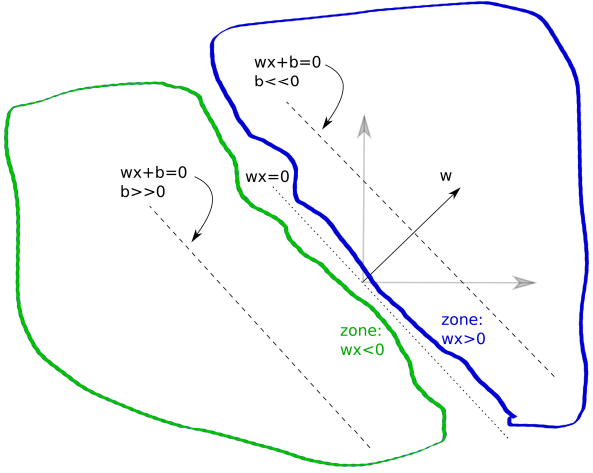
We know: $w \cdot x > 0$ for all points x that are on that side of the **hyperplane through the origin**, in which w points to.

The same also holds for 3 or more dimensions. All the vectors below solve $wx + b = 0$ for some bias $b < 0$!



In above figure x_1,x_2,x_3 all have $w \cdot x_i > 0$

The bias b shifts the hyperplane corresponding to $w \cdot x + b = 0$ parallel/anti-parallel to the direction of w.



the hyperplane is parallel to the hyperplane orthogonal to w which goes through the origin

The set of points x such that

$$\{x : \ w \cdot x + b = 0\}$$

for $b < 0$ is a hyperplane which is parallel to the hyperplane $\{x : x \cdot w = 0\}$ orthogonal to w going through the origin, and which is shifted towards the direction of w

The set of points x such that

$$\{x : \ wx + b = 0\}$$

for $b > 0$ is a hyperplane which is parallel to the hyperplane $\{x : x \cdot w = 0\}$ orthogonal to w going through the origin, and which is shifted opposite to the direction of w

hyperplane dependency on bias b

- $\{x : \ w \cdot x = 0\}$ is a hyperplane orthogonal to w, which contains the zero vector $(0, 0, \ldots, 0)$.
- Negative $b < 0$ shift the hyperplane $\{x : wx + b = 0\}$ towards the direction of w,
- positive $b > 0$ shift the hyperplane $\{x : wx + b = 0\}$ against the direction of w.
- Large absolute values $|b|$ shift it far away.

hyperplane explicit

The linear mapping $f(x) = w \cdot x + b$ is zero for x which consist of the plane of points

$$\left\{ x : x = u + -b\frac{w}{\|w\|^2}, u \text{ such that } w \cdot u = 0 \right\}$$

In this representation:

- u such that $w \cdot u = 0$ is the hyperplane of vectors u orthogonal to w.
- the vector $-b\frac{w}{\|w\|^2}$ shifts the hyperplane in direction of $\pm w$.

That holds because $w \cdot u = 0$ and $w \cdot w = \|w\|_2^2$:

$$\begin{aligned} w \cdot x + b &= w \cdot \left(u + -b\frac{w}{\|w\|^2} \right) + b \\ &= \underbrace{w \cdot u}_{=0} - b\frac{w \cdot w}{\|w\|^2} + b = 0 - b + b = 0 \end{aligned}$$

By subtracting its component parallel to w!

Be x any vector. Subtract its component parallel to w.

Important: use length-normalized w: $\frac{w}{\|w\|}$

$\left(x \cdot \frac{w}{\|w\|}\right)$ is the component of x in direction of $\frac{w}{\|w\|}$

$$u = x - \left(x \cdot \frac{w}{\|w\|}\right) \frac{w}{\|w\|} = x - (x \cdot w) \frac{1}{\|w\|^2} w$$

$$\Rightarrow u \cdot w = 0$$

cf. QR-decomposition to find an orthonormal basis of such vectors.

What is the set of points x where the prediction is a constant, that is $g_{w,b}(x) = c$?

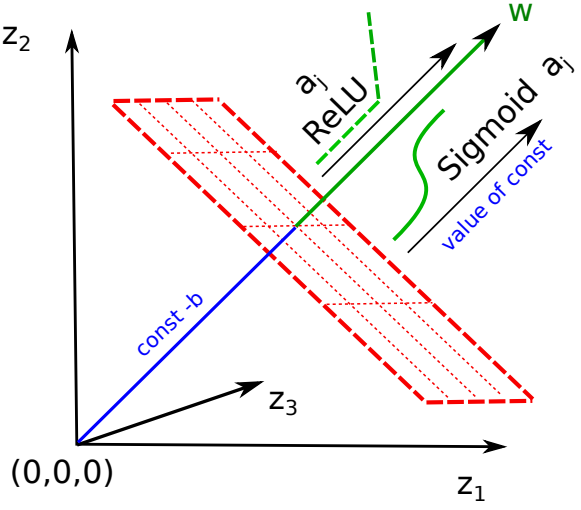
Answered by reducing it to a zero set:

$$g_{w,b}(x) = wx + b = c$$
$$wx + (b - c) = 0$$

The set of points x such that $g_{w,b}(x) = c$ is just the set $x : \ g_{w,b-c}(x) = 0$.

The points x such that $w \cdot x + b = c$ is a hyperplane orthogonal to w, shifted in direction of w by the amount of $(c - b)\frac{w}{\|w\|^2}$.
given this, the next slide should be clear:

in short:



- The weights w defines the orientation of the plane.
- The bias b defines the position of the plane.

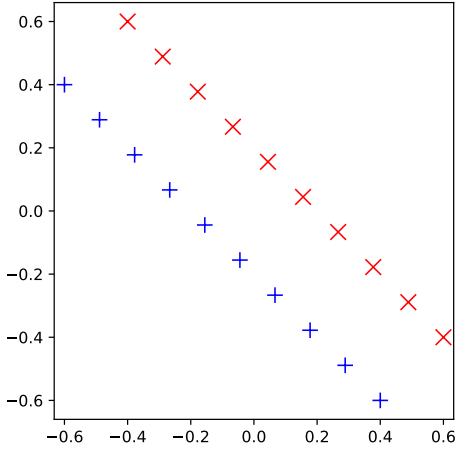
thinking task

What is the set of points $x = (x_1, x_2) \in \mathbb{R}^2$ such that

$$3x_1 - 2x_2 + 3 = 0 \ ?$$

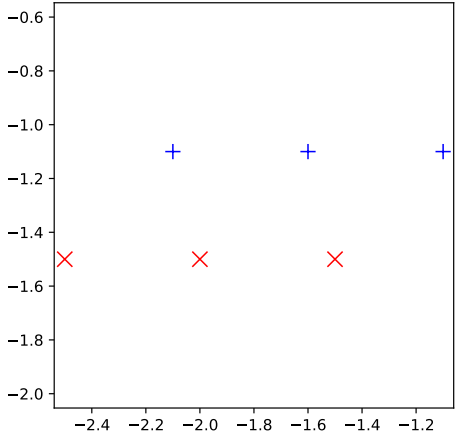
What is the set of points $x = (x_1, x_2, x_3) \in \mathbb{R}^3$ such that

$$x_1 - x_2 - 2x_3 + 2 = 0 \ ?$$



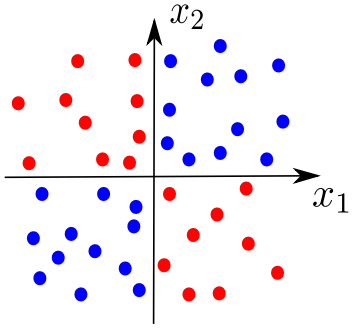
What would be a separating hyperplane in this case ?

You should be able to solve that now

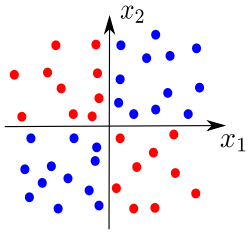


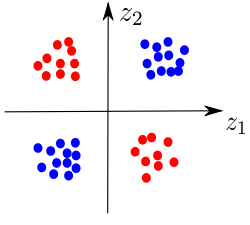
What would be a separating hyperplane in this case ?

To illustrate how concatenation of neurons allows learning non-linear mappings, consider separating red from blue samples in the following example:



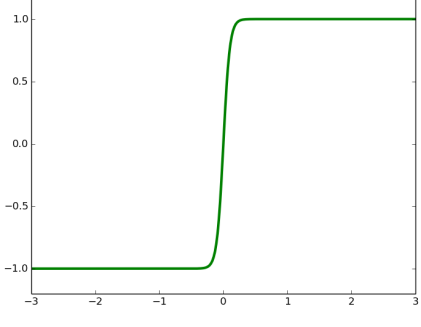
Samples lie in quadrants around coordinates $(\pm 1, \pm 1)$.

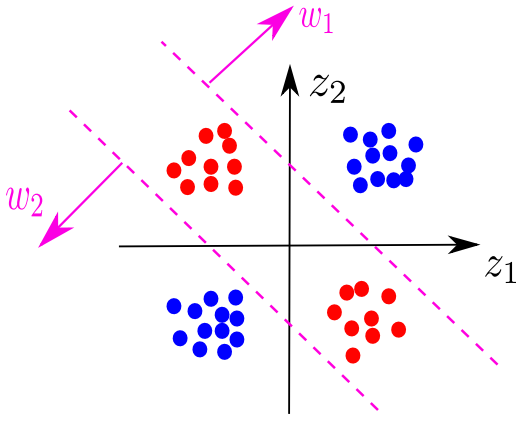




$$z_1 = tanh(10x_1)$$
$$z_2 = tanh(10x_2)$$

- *tanh* pushes points towards $(\pm 1, \pm 1)$

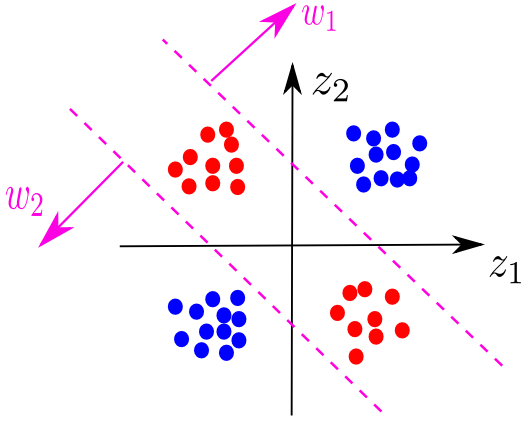




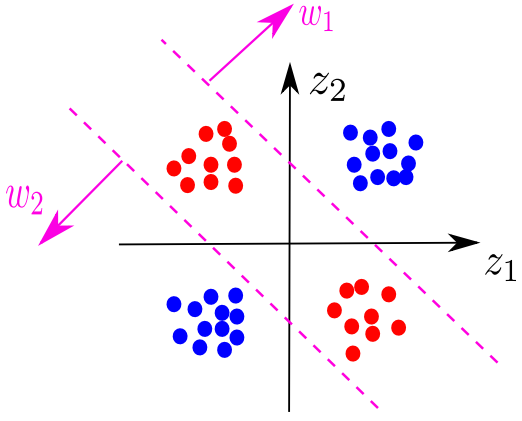
Idea:

- For proper choice of w_1 and b_1, $g_1 = \sigma(w_1 z + b_1)$ has high values ≈ 1 in the upper right and low values ≈ 0 elsewhere.
- For proper choice of w_2 and b_2, $g_2 = \sigma(w_2 z + b_2)$ has high values ≈ 1 in the lower left and low values ≈ 0 elsewhere.

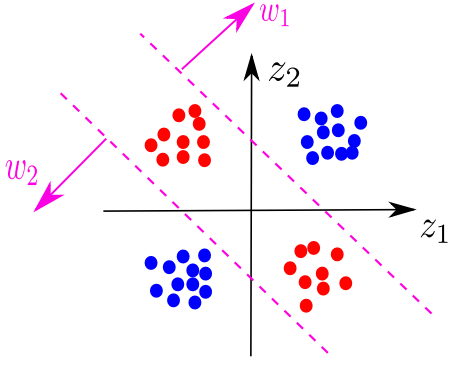
- Then, $g_1 + g_2$ will be ≈ 1 in the upper right and lower left corners.
- In the middle zone, $g_1 + g_2$ will be ≈ 0.



- $w_1 = (?, ?)$, $w_2 = (?, ?)$



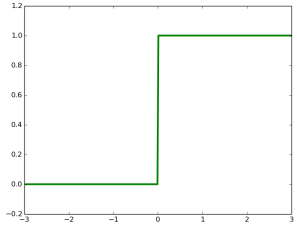
- If $w_1 = (1, 1)$, $w_2 = (-1, -1)$, then $w_1 \cdot z \approx 0$, $w_2 \cdot z \approx 0$ for the middle zone, while $w_1 \cdot z \gg 0$ in the upper right corner and $w_2 \cdot z \gg 0$ in the lower left corner.
- Could then choose $b_1 < 0$ and $b_2 < 0$ suitably.
- then for the middle zone: both $g_1(z) \approx 0$ and $g_2(z) \approx 0$
  (exact 0 if using hard thresholding neuron)
- outer zones: one of the g_1, g_2 will be ≈ 1
- Final output: $f(x) = \sigma(c(g_1 + g_2) + cb_3)$



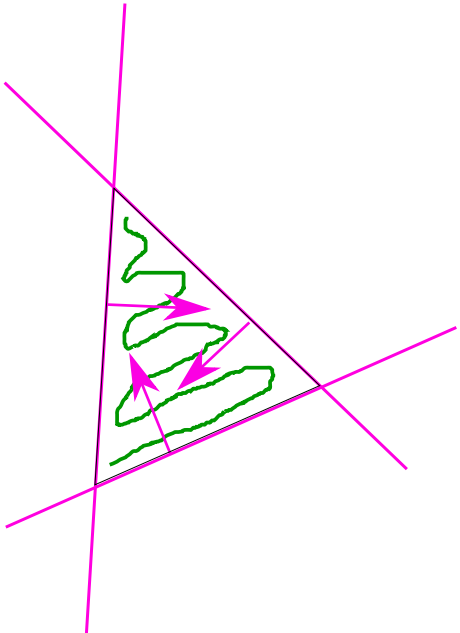
- Then, $g_1 + g_2 \approx 1$ in the upper right and lower left corners.
- In the middle zone, $g_1 + g_2 \approx 0$
- add a simple third layer with weights 1,1 and bias -0.5:

$$y = \mathit{thresh}(g_1 + g_2 - 0.5)$$

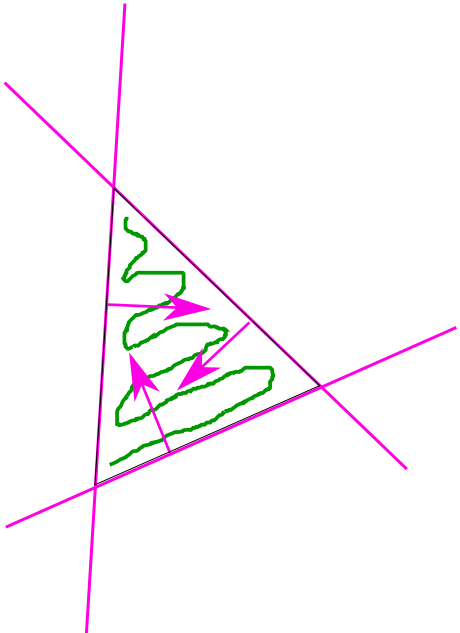
- Use threshold activation (for simplicity):



- A 2-layer network with n neurons in the hidden layer, and a thresholded sum in the second layer can represent any convex polygonal shape with n edges.
- With $n \to \infty$ neurons in the first layer, any convex shape can be **approximately** encoded.



- Adding a third layer allows the neural network to approximately encode any union of convex shapes.
- However, being able to (approximately) represent does not imply that this would be the result when learning from finite data.



Universal approximation theorem (a version)

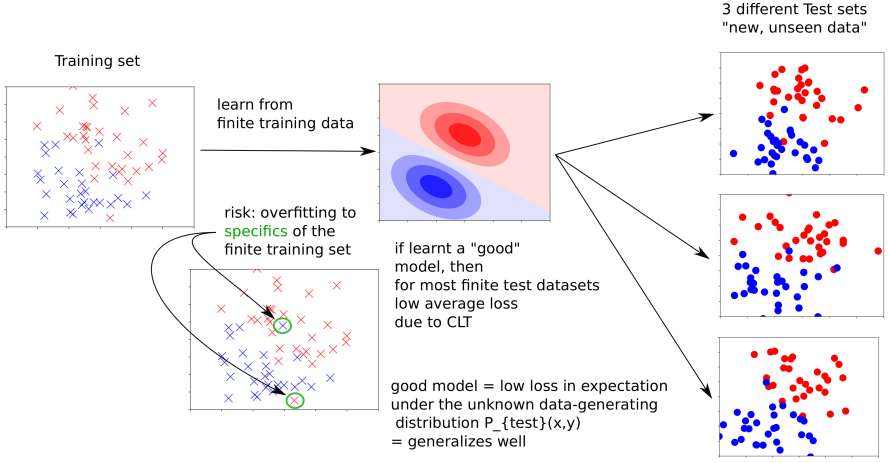
Let $g(\cdot)$ be a continuous function on a m-dimensional hypercube $[0,1]^m$. Let $a(\cdot)$ be a non-constant, bounded, continuous (activation) function. Then $g(\cdot)$ can be approximated arbitrarily well, that is, for every maximal deviation $\epsilon > 0$, there exists a set of weights u_i, w_i and biases b_i such that:

$$\forall x \in [0,1]^m : \ |g(x) - (\sum_{i=1} u_i a(w_i x + b_i) + b)| < \epsilon$$

Neural networks with one hidden layer and two layers of weights is able to approximate any smooth function on a compact hypercube. If there is a good algorithm for learning the parameters from finite data, we are done!

- Universal approximation theorem was misleading for neural network research in the 90s.
- Kolmogorov (1957), Hornik (1989), Cybenko (1989) and others: Neural network is able to approximate *any continuous function on a compact region*.

- Kolmogorov (1957), Hornik (1989), Cybenko (1989) and others: Neural network is able to approximate *any continuous function on a compact region*.
- However, ability to approximate any function $\neq$ ability to learn any function well from finite training data (... overfitting)



Right because you can represent any shape, you will easily reserve a blue area for the blue point in the training data on the wrong side. This results in overfitting. need to restrict models to prevent this!!!

- convolution layers as locally restricted linear operators
- regularizations on gradient flow