Deep Learning

Ulf Brefeld & Soham Majumder

build: April 8, 2024

Machine Learning Group
Leuphana University of Lüneburg

Supervised Machine Learning

Goal: find a deterministic mapping from a set of inputs X to a set of target variables Y

The relation between X and Y is given by a (generally unknown) joint probablity distribution $\P XY$

If $\P XY$ was known, we could use

$$\P Y|X = x(y) = \frac{\P XY(x, y)}{\sum_{\tilde{y} \in Y} \P XY(x, \tilde{y})} = \frac{\P XY(x, y)}{\P X(x)}$$

Supervised learning:

Instead of finding a model for $\P XY$, learn a model for $\P Y|X = x(y)$ directly

Learn a function $f\colon X \to Y$ that has a low generalization error for new and unseen $x \in X$

The **generalization error** (theoretical risk) is the expected loss

$$R[f] = \int_{X \times Y} V(x, y, f) d\P XY$$

where $V : X \times Y \times F \to \mathbb{R}_0^+$ is a loss function

BUT: joint probablity $\P XY$ is unknown and $\inf_f R[f]$ cannot be computed directly

Solution: Approximate $R[f]$ using an N-sample (training set)

$$\mathcal{D} = \{(x_n, y_n)\}_{1 \le n \le N} \in X \times Y$$

drawn **independent and identically distributed (iid)** from $\P XY$

Empirical Risk Minimization

Empirical Risk Minimization

Idea: minimize the **empirical risk** on $\mathcal{D}$

$$\hat{R}[f] = \frac{1}{N} \sum_{n=1}^{N} V(x_n, y_n, f)$$

Converges asymptotically to $R[f]$ in the limit $N \to \infty$

Drawbacks:

- Convergence $\hat{R}[f] \to R[f]$ slow, requires large N
- In general no unique minimum (possibly poor generalization, no distinguished solution)

Tensors

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

- A 1d tensor is a vector (e.g. a sound sample),
- A 2d tensor is a matrix (e.g. a grayscale image),
- A 3d tensor is a vector of identically sized matrices (e.g. a multi-channel image),
- A 4d tensor is a matrix of identically sized matrices (e.g. a sequence of multi-channel images),
- etc.

Tensors are used to encode the signal to process, but also the internal states and parameters of the *neural networks*.

Manipulating data through this constrained structure allows to use CPUs and GPUs at peak performance.

Compounded data structures can represent more diverse data types.

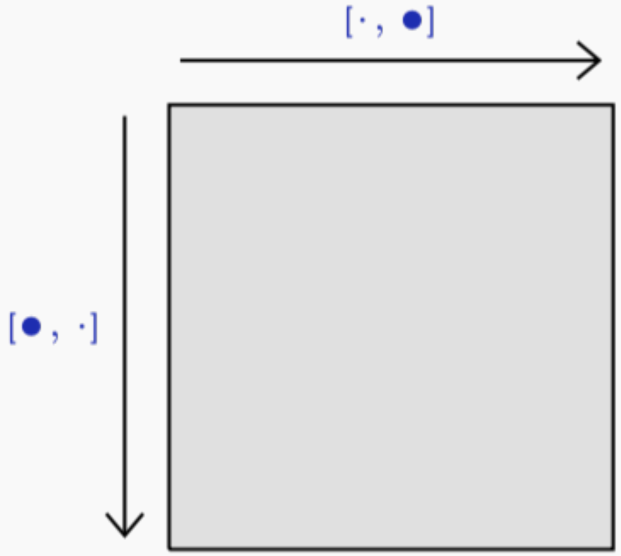
In a nutshell, tensor multiplications allow to express equations like these in a single line of code:

$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data}\in\mathbb{R}^{N\times 2}} \qquad \underbrace{\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix}}_{x\in\mathbb{R}^{N\times 2}} \underbrace{\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}}_{w\in\mathbb{R}^{2\times 1}} \approx \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}_{y\in\mathbb{R}^{N\times 1}}$$

GPUs are optimized for processing tensor multiplications.

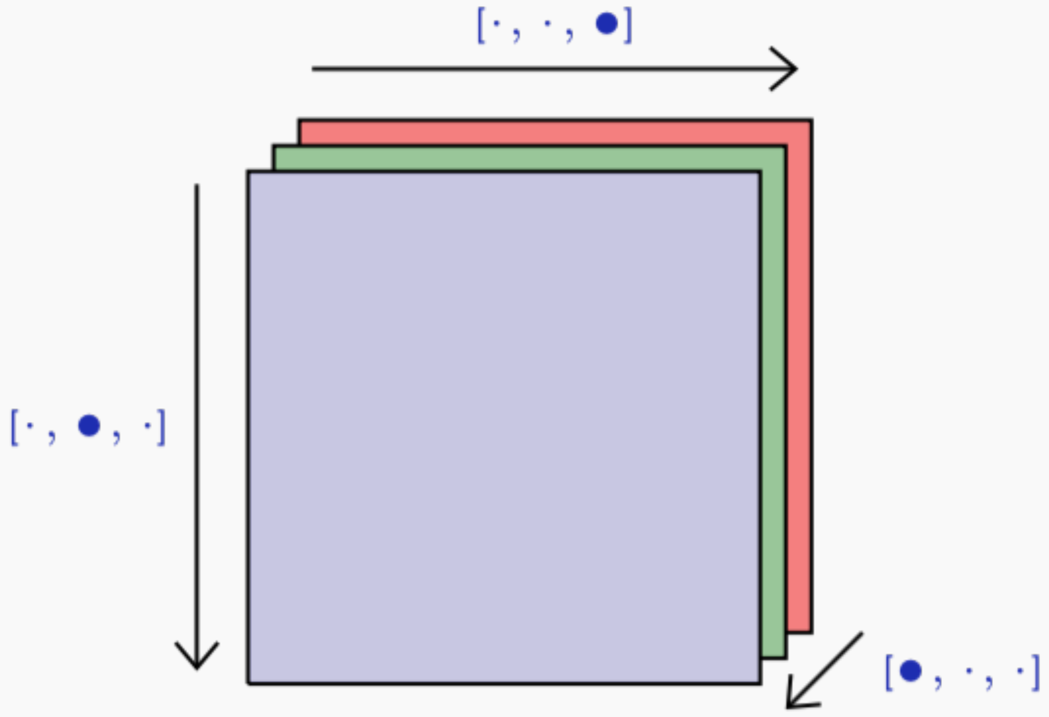
Example

2d tensor (*e.g.* grayscale image)



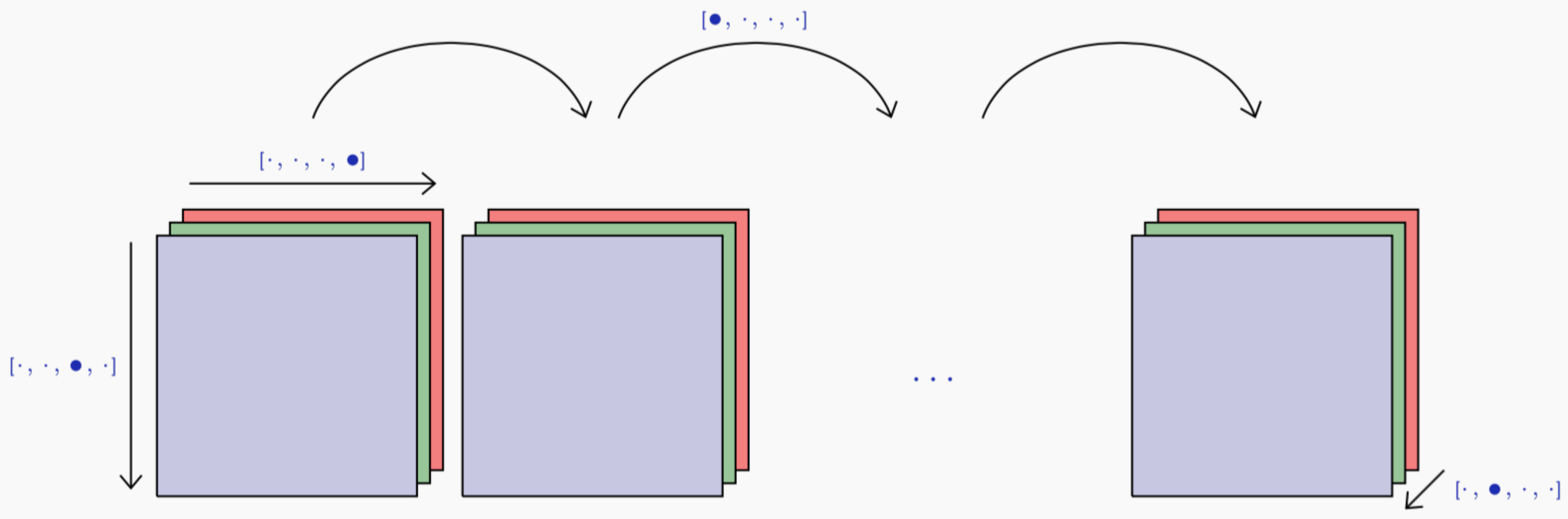
Example

3d tensor (*e.g.* rgb image)


$[\cdot, \cdot, \bullet]$
$[\cdot, \bullet, \cdot]$
$[\bullet, \cdot, \cdot]$

Example

4d tensor (*e.g.* sequence of rgb images)



Perceptrons

Threshold Logic Unit (McCulloch & Pitts, 1943)

The first mathematical model for a neuron was the Threshold Logic Unit, with Boolean inputs and outputs: $f(x) = \mathbf{1}_{\{\sum_i w_i x_i + b \geq 0\}}$.

It can in particular implement

$$\begin{aligned} \text{or}(u, v) &= \mathbf{1}_{\{u+v-\frac{1}{2}\geq 0\}} & w = [1, 1], b = -\frac{1}{2} \\ \text{and}(u, v) &= \mathbf{1}_{\{u+v-\frac{3}{2}\geq 0\}} & w = [1, 1], b = -\frac{3}{2} \\ \text{not}(u) &= \mathbf{1}_{\{-u+\frac{1}{2}\geq 0\}} & w = [-1], b = \frac{1}{2} \end{aligned}$$

Hence, **any Boolean function** can be build with such units

Perceptrons (Rosenblatt, 1957)

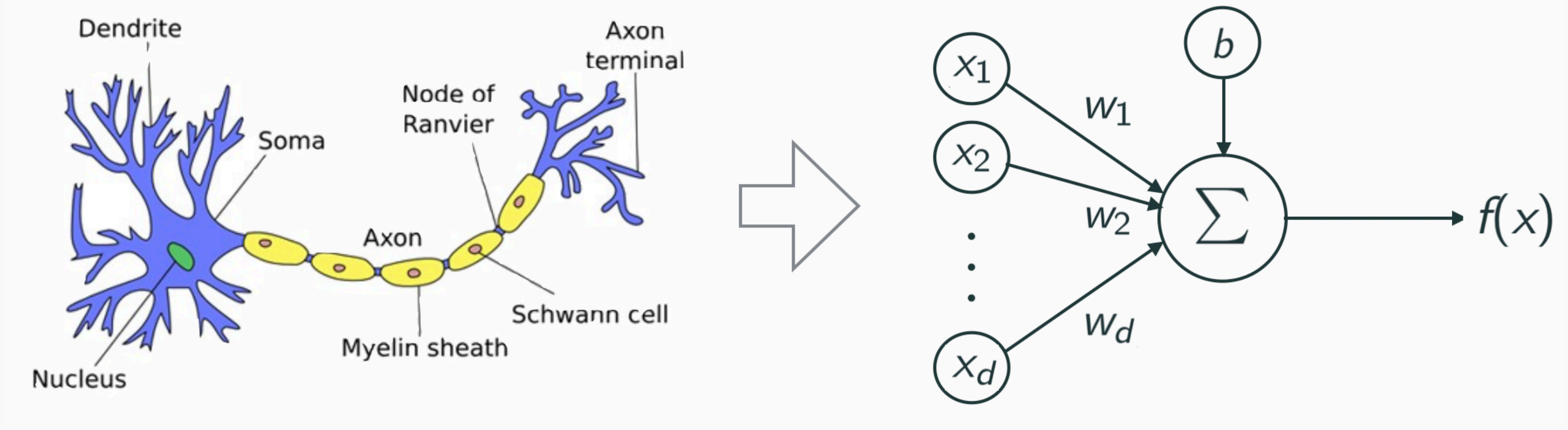
The perceptron is very similar

$$f(x) = \begin{Bmatrix} 1 & : & \sum_i w_i x_i + b \geq 0 \\ 0 & : & otherwise \end{Bmatrix}$$

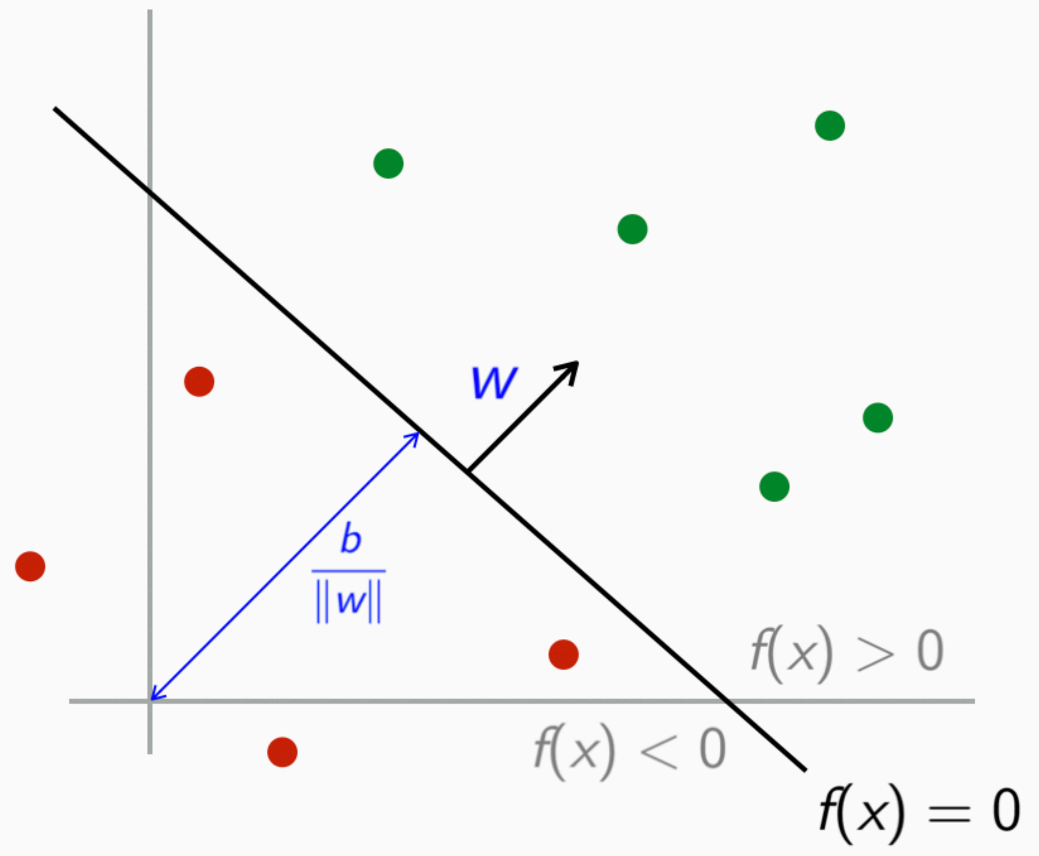
but the inputs are real values and the weights can be different.

This model was originally motivated by biology, with w_i being the synaptic weights, and x_i and f firing rates. It is a (very) crude biological model.

The perceptron was originally motivated by biology, with w_d being the synaptic weights, inputs x_d and firing rates f. It is a (very) crude biological model:



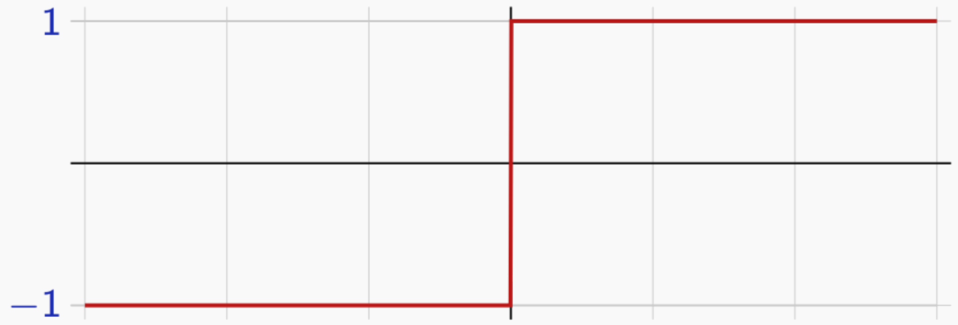
The perceptron realizes a hyperplane in input space with normal vector w and distance $b/\|w\|$ from the origin



Instances of the positive class should lie in the positive halfspace, instances of the negative class in the negative halfspace, respectively

To make things simpler, we take responses $y \in \{\pm 1\}$. Let

$$\sigma(z) = \begin{Bmatrix} 1 & : & z \geq 0 \\ -1 & : & otherwise \end{Bmatrix}$$



The perceptron classification rule boils down to

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

For neural networks, the function σ that follows a linear operator is called the **activation function**

Relation to Empirical Risk Minimization

The loss function of the perceptron is given by

$$V_{perc}(x_n, y_n, f) = -\min\left(0, y_n f(x_n)\right)$$

The perceptron is directly updated after every mistake. We refer to such updates as *stochastic gradient descent*. The gradients are given by

$$\frac{\partial V_{perc}}{\partial w} = \begin{Bmatrix} -y_n x_n & : & y_n f(x_n) < 0 \\ 0 & : & y_n f(x_n) \geq 0 \end{Bmatrix}$$

and $\partial V_{perc}/\partial w$ analogously in case b is not augmented in w.

Recall: Batch vs Online Learning

Batch learning uses the full gradient (batch), that is, we run over the entire training set and collect all gradients and perform a single update step using the **true gradient** afterwards:

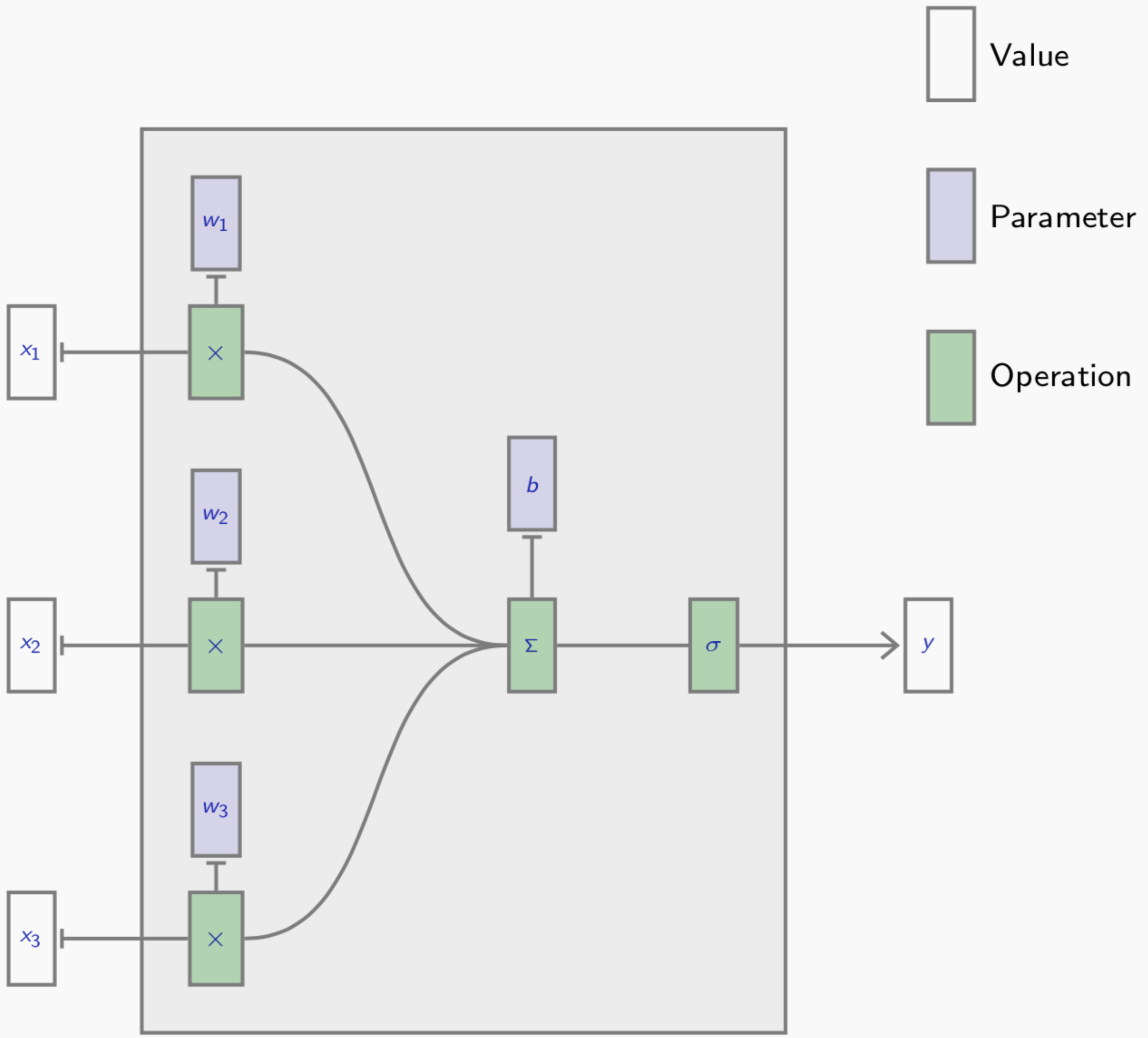
$$\frac{\partial}{\partial w} \sum_{n=1}^{N} V(x_n, y_n, f)$$

Online learning updates the model after every instance (or mistake) directly. We use the loss function to compute the gradient

$$\frac{\partial}{\partial w} V(x_n, y_n, f).$$

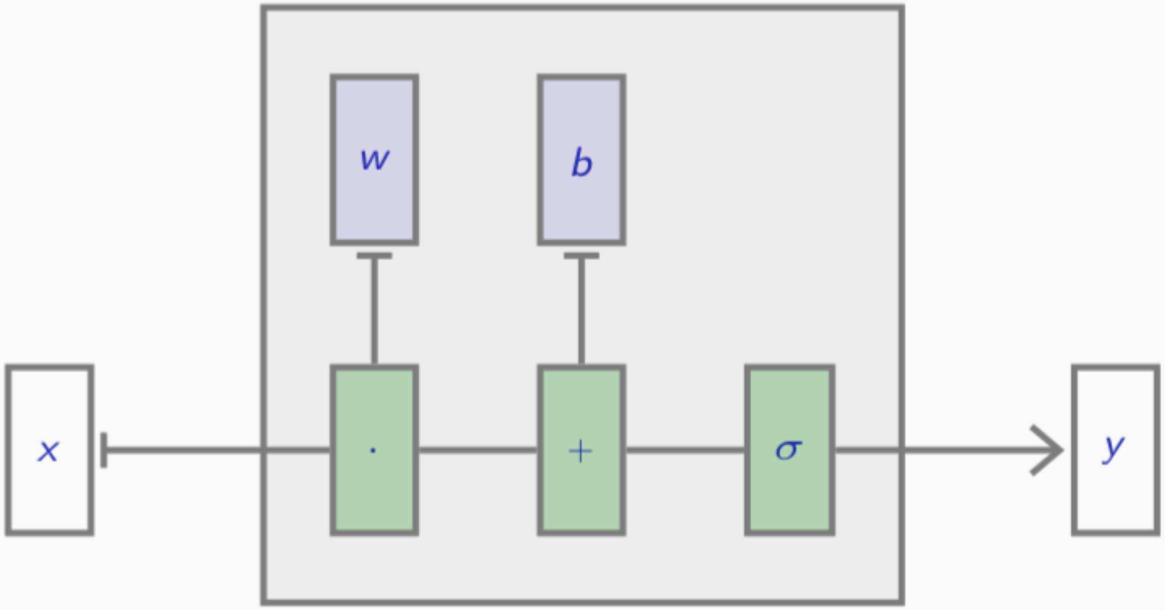
Due the (random) ordering of the training instances, this scheme resembles a stochastic process and is called **stochastic gradient descent (SGD)**

We can represent this *neuron* $f(x) = \sigma(w^\top x + b)$ as follows:



In tensor notation, the equation/figure is much simpler:

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



Later, the input can be any multi-dimensional tensor, the dimensions of w and b are directly defined by the size of x

Given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{-1, +1\}, n = 1, \ldots, N,$$

a very simple scheme to train such a linear operator for classification is the perceptron algorithm:

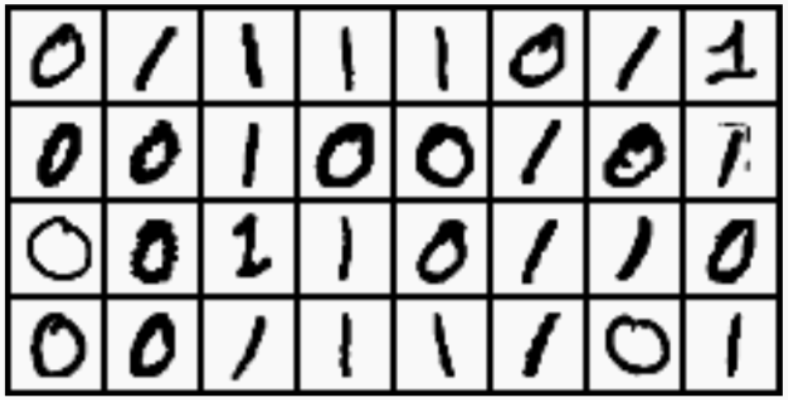
$w = (b, w_1, w_2, \ldots)$
$x = (1, x_1, x_2 \ldots)$

1. Start with $w^0 = \mathbf{0}$
2. while $\exists n_k$ s.t. $y_{n_k}\left(w^k \cdot x_{n_k}\right) \leq 0$, update $w^{k+1} = w^k + y_{n_k} x_{n_k}$

exists

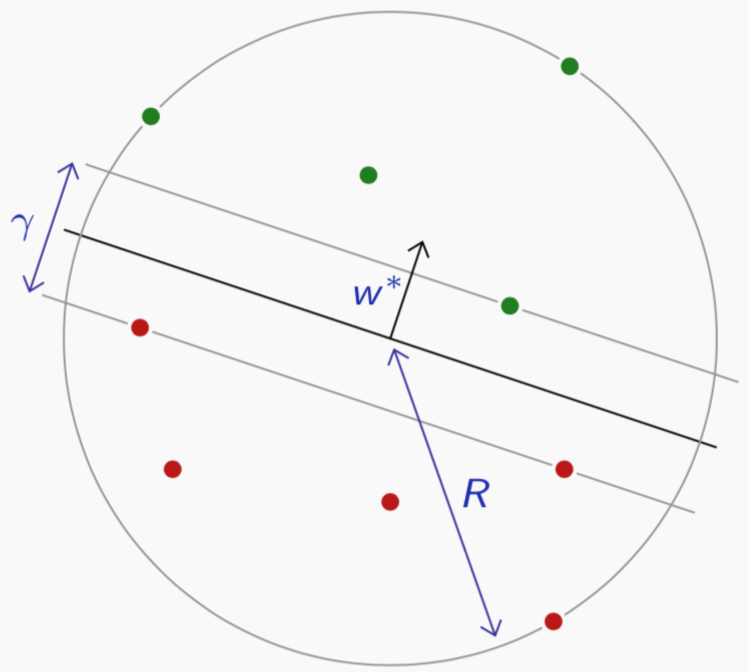
The bias b can be introduced as one of the w's by adding a constant component to x equal to 1.

This crude algorithm works often surprisingly well. With MNIST's '0's as negative class, and '1's as positive one.



```
epoch 0 nb_changes 64 train_error 0.23% test_error 0.19%
epoch 1 nb_changes 24 train_error 0.07% test_error 0.00%
epoch 2 nb_changes 10 train_error 0.06% test_error 0.05%
epoch 3 nb_changes 6 train_error 0.03% test_error 0.14%
epoch 4 nb_changes 5 train_error 0.03% test_error 0.09%
epoch 5 nb_changes 4 train_error 0.02% test_error 0.14%
epoch 6 nb_changes 3 train_error 0.01% test_error 0.14%
epoch 7 nb_changes 2 train_error 0.00% test_error 0.14%
epoch 8 nb_changes 0 train_error 0.00% test_error 0.14%
```

We can get a convergence result under two assumptions:



1. The x_n are in a sphere of radius R:

$$\exists R > 0 : \forall n \ \ \|x_n\| \leq R$$

2. The two populations can be separated with a margin $\gamma > 0$:

$$\exists w^* > 0, \|w^*\| = 1, \exists \gamma > 0 : \forall n \ \ y_n(w^* \cdot x_n) \geq \gamma/2$$

To prove the convergence, let us make the assumption that there still is a misclassified sample at iteration k, and w^{k+1} is the weight vector updated with it. We have

$$
\begin{aligned}
w^{k+1} \cdot w^* &= \left(w^k + y_{n_k} x_{n_k}\right) \cdot w^* \\
&= w^k \cdot w^* + \underbrace{y_{n_k} (x_{n_k} \cdot w^*)} \\
&\geq w^k \cdot w^* + \gamma/2 \\
&\geq (k+1)\gamma/2
\end{aligned}
$$

The Cauchy-Schwarz inequality states

$$
\|w^k\| \underbrace{\|w^*\|}_{=1} \geq w^k \cdot w^*
$$

and we get

$$
\begin{aligned}
\|w^k\|^2 &\geq \left(w^k \cdot w^*\right)^2 / \|w^*\|^2 \\
&\geq k^2\gamma^2/4
\end{aligned}
$$

And

$$\begin{aligned}
\|w^{k+1}\|^2 &= w^{k+1} \cdot w^{k+1} \\
&= \left(w^k + y_{n_k} x_{n_k}\right) \cdot \left(w^k + y_{n_k} x_{n_k}\right) \\
&= w^k \cdot w^k + \cancel{\underbrace{2 y_{n_k} w^k \cdot x_{n_k}}_{\leq 0}} + \underbrace{\|x_{n_k}\|^2}_{\leq R^2} \\
&\leq \|w^k\|^2 + R^2 \\
&\leq (k+1) R^2
\end{aligned}$$

Putting these two results together, we get

$$k^2\gamma^2/4 \leq \|w^k\|^2 \leq kR^2$$

hence

$$k \leq 4R^2/\gamma^2$$

That is, no misclassified sample can remain after $\lfloor 4R^2/\gamma^2 \rfloor$ iterations.

This result makes sense:

1. The bound does not change if the population is scaled, and
2. the larger the margin, the more quickly the algorithm classifies all the samples correctly.

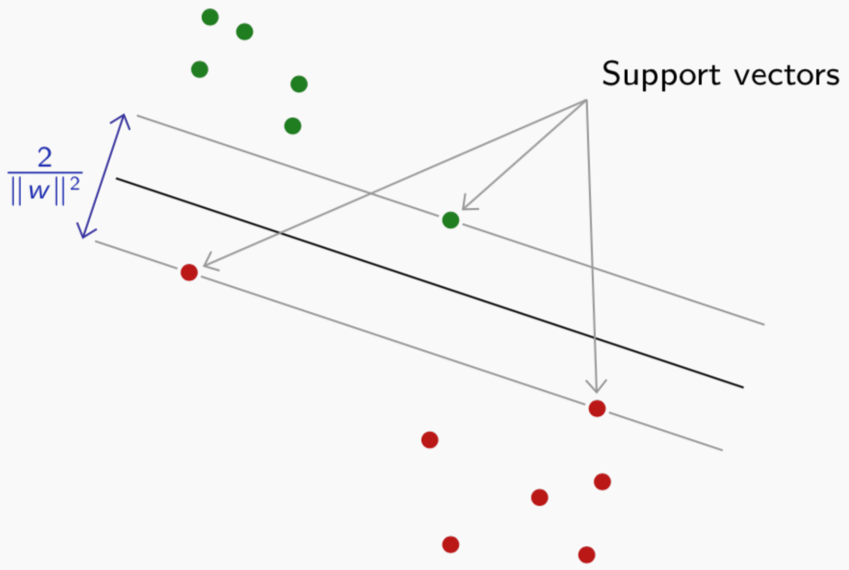
The perceptron stops as soon as it finds a separating boundary.

Other algorithms maximize the distance of samples to the decision boundary, which improves robustness to noise. Support Vector Machines (SVM) achieve this by minimizing

$$L(w, b) = \frac{1}{2}\|w\|^2 + \frac{C}{N}\sum_n \max\left(0, 1 - y_n(w \cdot x_n + b)\right),$$

which is convex and has a global optimum.

$$L(w, b) = \frac{1}{2}\|w\|^2 + \frac{C}{N}\sum_n \max\left(0, 1 - y_n(w \cdot x_n + b)\right),$$



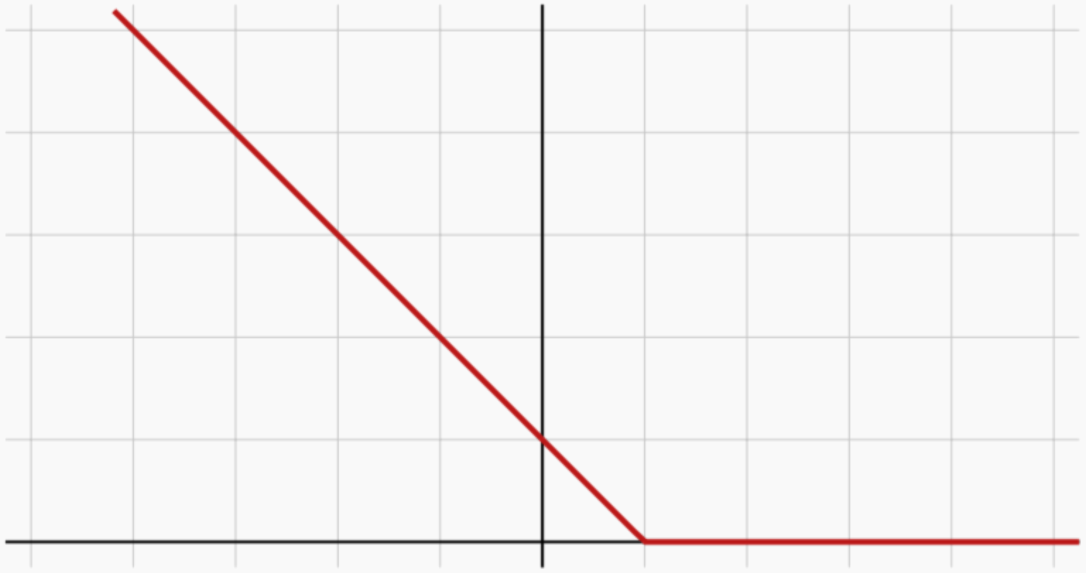
Minimizing $\max\left(0, 1 - y_n(w \cdot x_n + b)\right)$ pushes the n-th sample beyond the plane $w \cdot x + b = y_n$, and minimizing $\|w\|^2$ increases the distance between the $w \cdot x + b = \pm 1$.

At convergence, only a small number of samples matter, the *support vectors*.

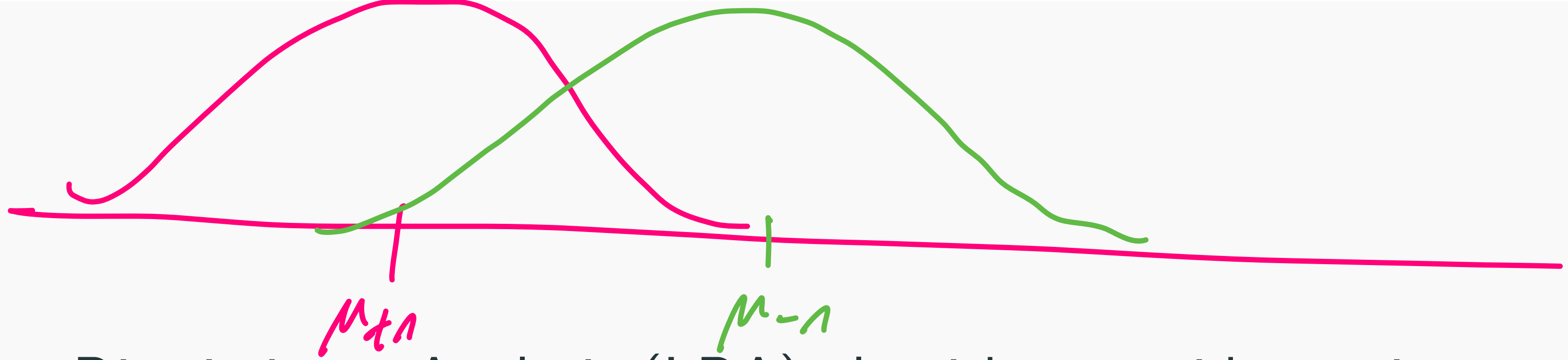
The term

$$V_{hinge}(f, x, y) = \max(0, 1 - yf(x))$$

is the so-called *hinge loss*.



Probabilistic interpretation of linear classifiers



The Linear Discriminant Analysis (LDA) algorithm provides a nice bridge between these linear classifiers and probabilistic modeling.

Consider the following class populations: $\forall y \in \{0,1\}, x \in \mathbb{R}^D$,

$$\mu_{X|Y=y}(x) = \overset{=b}{\boxed{\frac{1}{\sqrt{(2\pi)^D|\Sigma|}}}} \exp\left(-\frac{1}{2}(x - m_y)\Sigma^{-1}(x - m_y)^\top\right).$$

That is, they are Gaussian with the same covariance matrix Σ. This is the homoscedasticity assumption.

We have

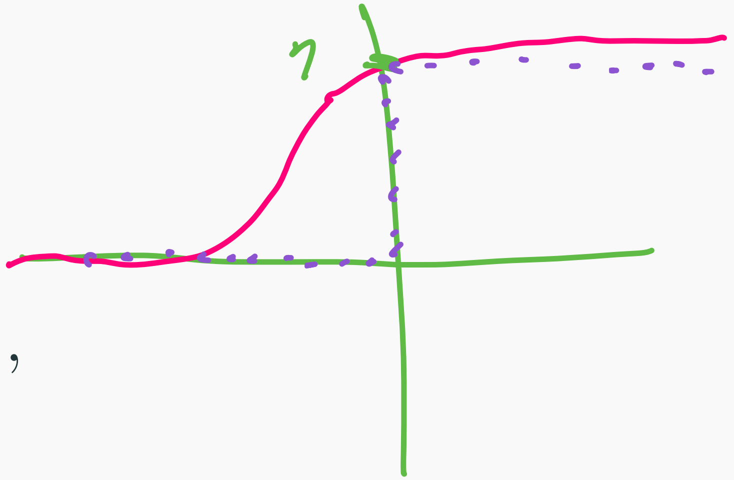
$$P(Y=1|X=x) = \frac{\overbrace{\mu_{X|Y=1}(x)}^{\text{likelihood}}\,\overbrace{P(Y=1)}^{\text{Prior}}}{\underbrace{\mu_X(x)}_{\text{Evidence}}}$$

$$= \frac{\mu_{X|Y=1}(x)P(Y=1)}{\mu_{X|Y=0}(x)P(Y=0) + \mu_{X|Y=1}(x)P(Y=1)}$$

$$= \frac{1}{1 + \frac{\mu_{X|Y=0}(x)}{\mu_{X|Y=1}(x)}\frac{P(Y=0)}{P(Y=1)}}$$

It follows that, with

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



we get

$$P(Y=1|X=x) = \sigma\left(\log\frac{\mu_{X|Y=1}(x)}{\mu_{X|Y=0}(x)} + \log\frac{P(Y=1)}{P(Y=0)}\right)$$

So with our Gaussian $\mu_X | Y = y$ of same Σ, we get

$$P(Y = 1 | X = x)$$

$$= \sigma\left(\log \frac{\mu_{X|Y=1}(x)}{\mu_{X|Y=0}(x)} + \underbrace{\log \frac{P(Y=1)}{P(Y=0)}}_{a} \right) \longrightarrow \text{constant}$$

$$= \sigma(\log \mu_{X|Y=1}(x) - \log \mu_{X|Y=0}(x) + a)$$

$$= \sigma\left(-\frac{1}{2}(x - m_1)\Sigma^{-1}(x - m_1)^\top + \frac{1}{2}(x - m_0)\Sigma^{-1}(x - m_0)^\top + a \right)$$

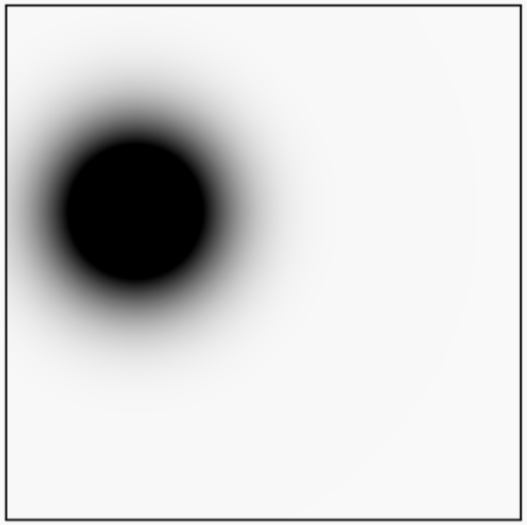
$$= \sigma\left(\cancel{-\frac{1}{2}x\Sigma^{-1}x^\top} + m_1\Sigma^{-1}x^\top - \frac{1}{2}m_1\Sigma^{-1}m_1^\top \right.$$

$$\left. \cancel{+\frac{1}{2}x\Sigma^{-1}x^\top} - m_0\Sigma^{-1}x^\top + \frac{1}{2}m_0\Sigma^{-1}m_0^\top + a \right)$$

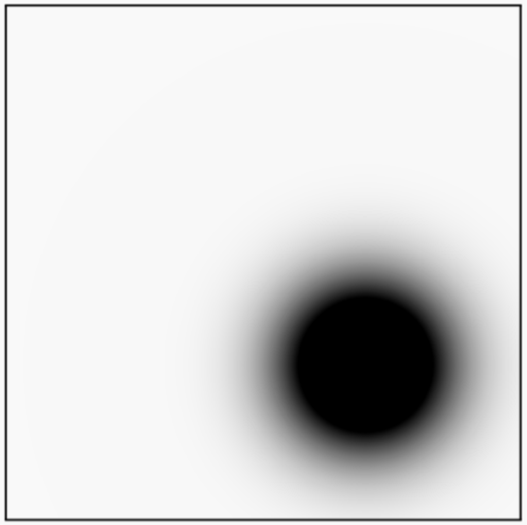
$$= \sigma\left((m_1 - m_0)\Sigma^{-1}x^\top + \underbrace{\frac{1}{2}\left(m_0\Sigma^{-1}m_0^\top - m_1\Sigma^{-1}m_1^\top \right) + a}_{\text{constant}} \right)$$

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

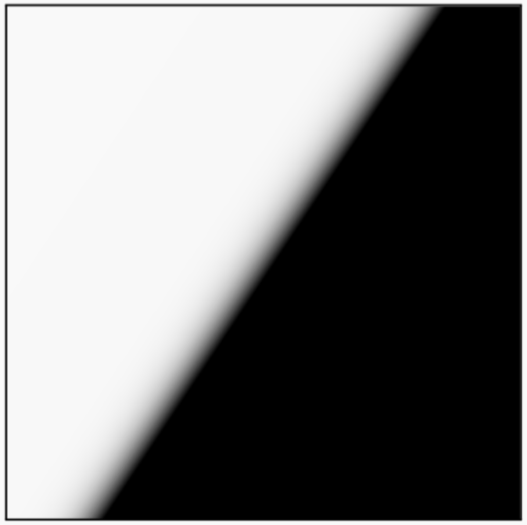
The homoscedasticity makes the second-order terms vanish.



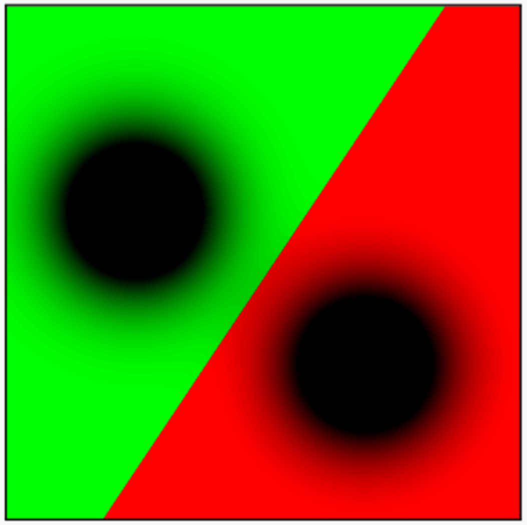
$\mu_{X|Y=0}$

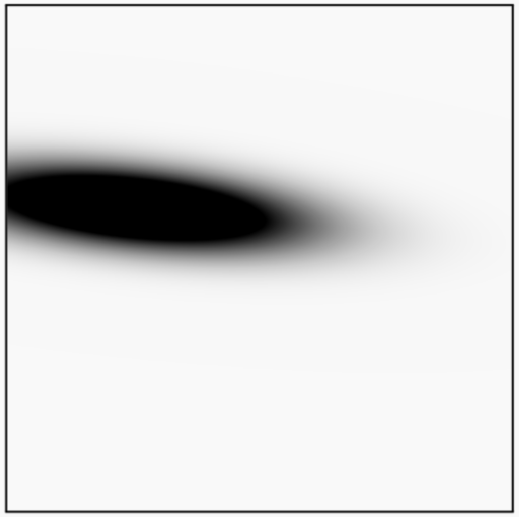


$\mu_{X|Y=1}$

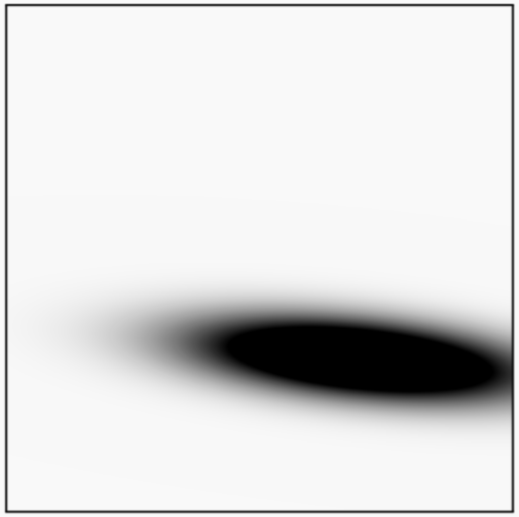


$P(Y = 1 \mid X = x)$

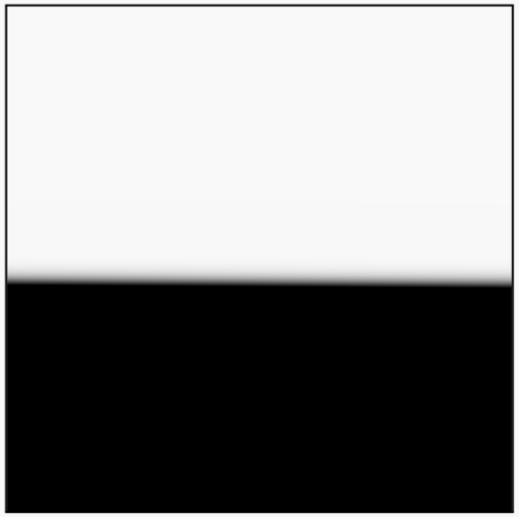




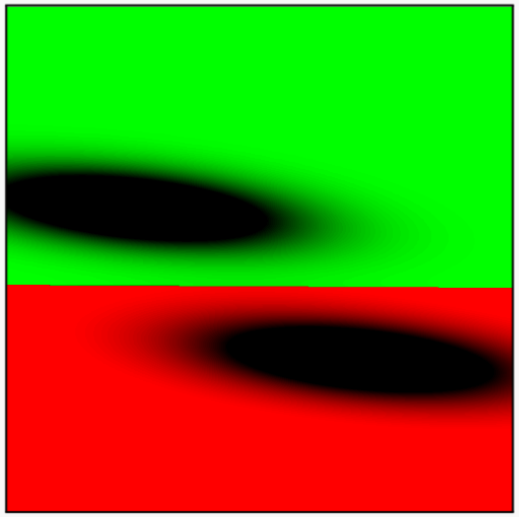
$\mu_{X \mid Y=0}$

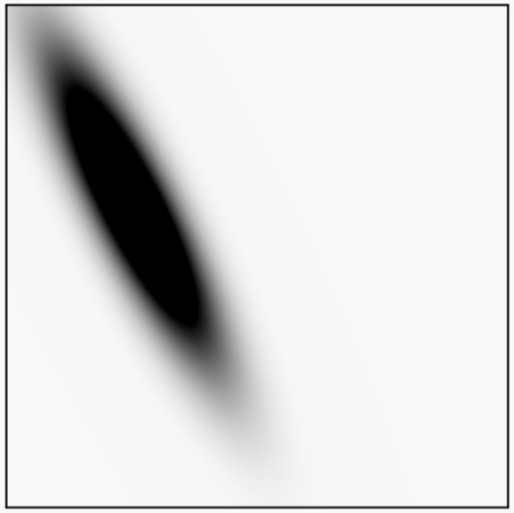


$\mu_{X \mid Y=1}$

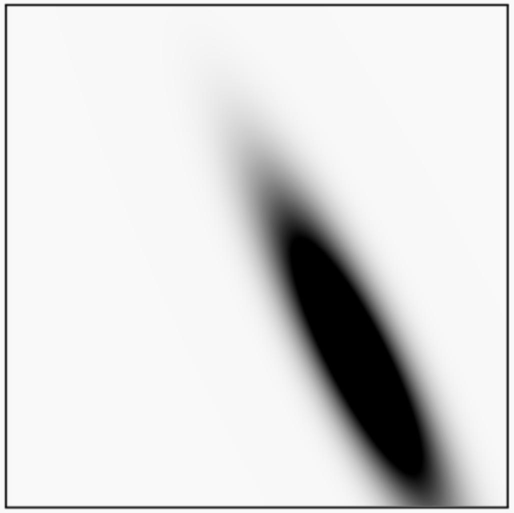


$P(Y = 1 \mid X = x)$

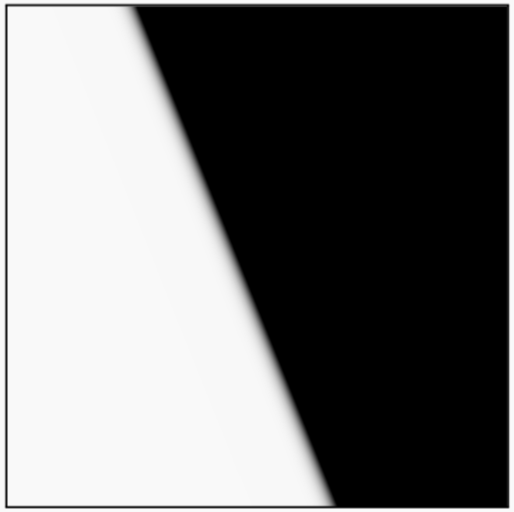




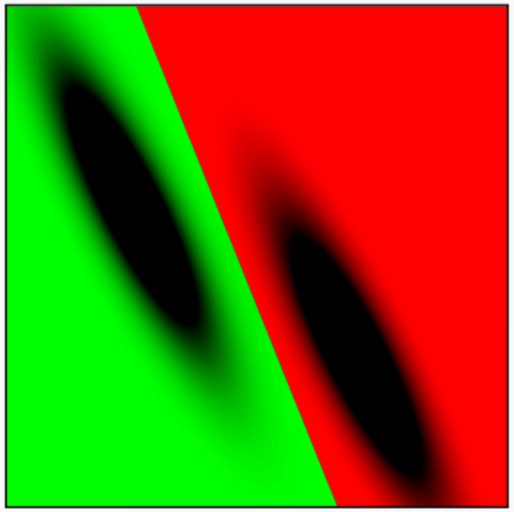
$\mu_{X|Y=0}$

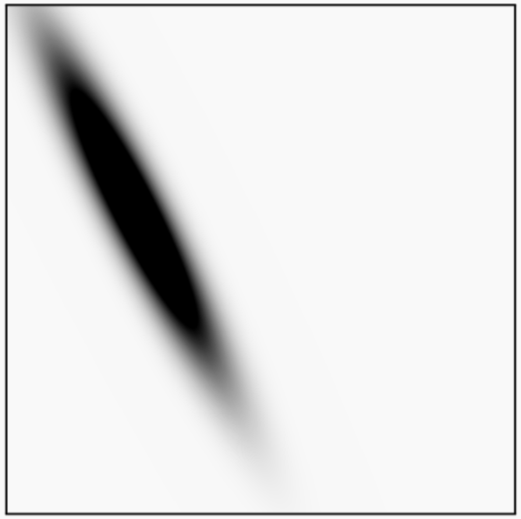


$\mu_{X|Y=1}$

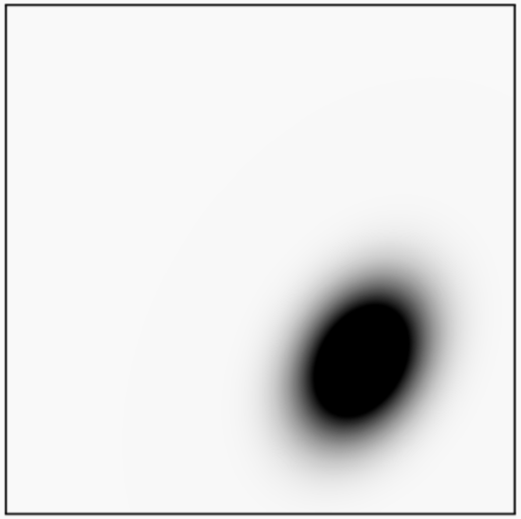


$P(Y = 1 \mid X = x)$

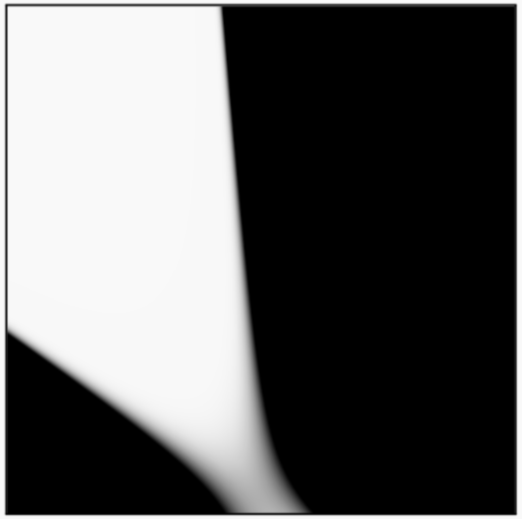




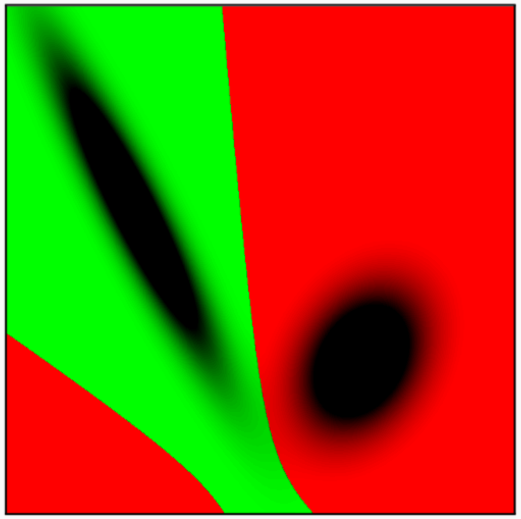
$\mu_{X \mid Y=0}$



$\mu_{X \mid Y=1}$



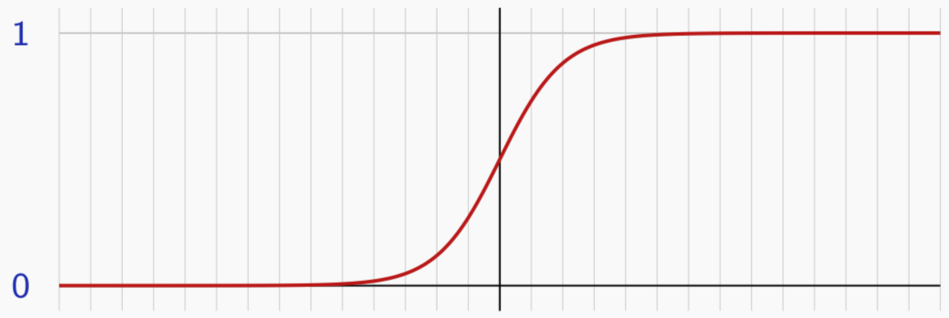
$P(Y = 1 \mid X = x)$



Note that the (logistic) sigmoid function

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

looks like a *soft heavyside* or step function



The overall model

$$f(x; w, b) = \sigma(w \cdot x + b)$$

looks very similar to the perceptron.

$P(x|w,b)\, P(w,b)$

We can use the model from LDA

$$P(w,b|X) = \frac{P(X|w,b)\,P(w,b)}{P(X)}$$

$$f(x; w, b) = \sigma(w \cdot x + b)$$

but instead of modeling the densities and derive the values of w and b, directly compute them by maximizing their probability given the training data.

First, to simplify the next slide, note that we have

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

hence, if Y takes values in $\{-1, +1\}$ then

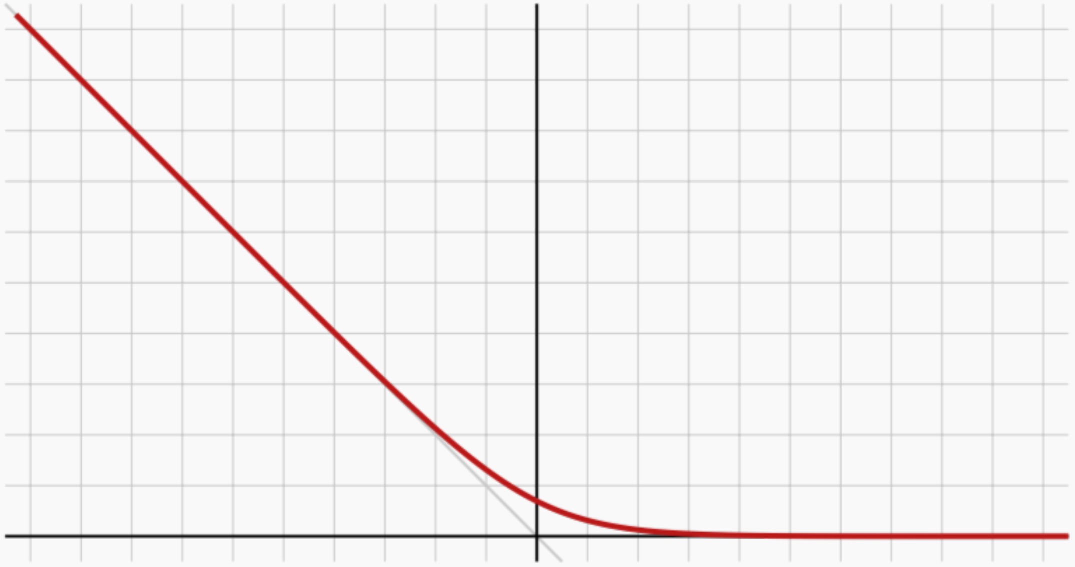
$$P(Y = y|X = x) = \underline{\sigma(y(w \cdot x + b))}$$

We have

$$
\begin{aligned}
&\log \mu_{W,B}(w, b | D = \mathcal{D}) \\
&= \log \frac{\mu_D(\mathcal{D} | W = w, B = b)\mu_{W,B}(w, b)}{\mu_D(\mathcal{D})} \\
&= \log \mu_D(\mathcal{D} | W = w, B = b) + \log \mu_{W,B}(w, b) - \log \cancel{Z}\ \mu_D(\mathcal{D}) \\
&= \sum_n \log \sigma(y_n(w \cdot x_n + b)) + \log \mu_{W,B}(w, b) - \log \cancel{Z}\ \mu_D(\mathcal{D})
\end{aligned}
$$

This is the logistic regression, whose loss aims at minimizing

$$-\log \sigma(y_n f(x_n))$$



Although the probabilistic and Bayesian formulations may be helpful in certain contexts, the bulk of deep learning is disconnected from such modeling.

We will come back sometime to a probabilistic interpretation, but most of the methods will be envisioned from the signal-processing and optimization angles.

Multi-dimensional output

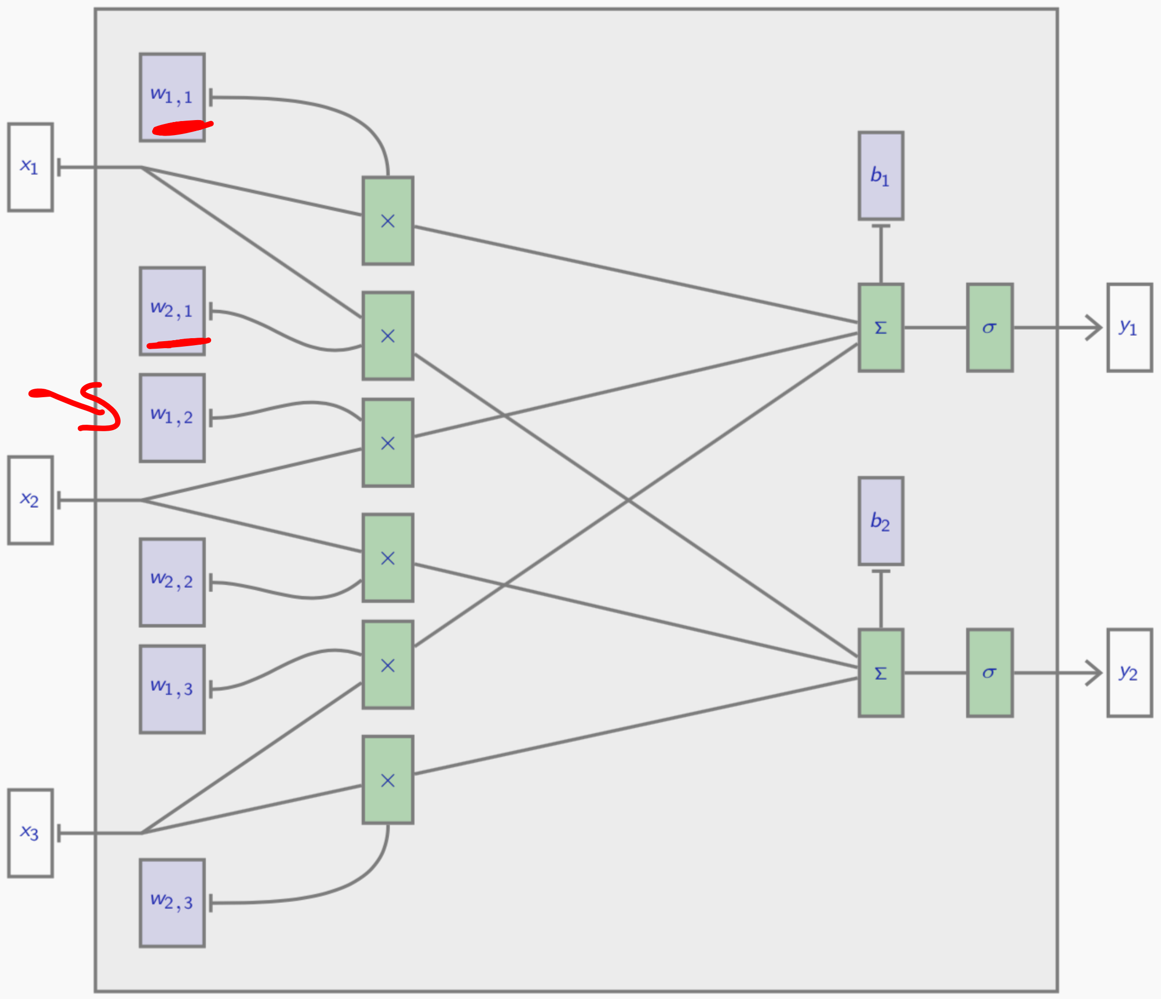
We can combine multiple linear predictors into a *layer* that takes D inputs and produces M outputs:

$$\forall \overset{m}{i} = 1, \ldots, M : \quad y_{\overset{m}{i}} = \sigma\left(\sum_{\overset{d}{j}=1}^{D} w_{\overset{md}{ij}} x_{\overset{d}{j}} + b_{\overset{m}{i}}\right)$$

where $b_{\overset{m}{\cancel{i}}}$ is the **bias** of the $\underset{m}{\cancel{i}}$-th unit, and $\underset{m}{w_{i1}}, \ldots, \underset{m}{w_{iD}}$ are its **weights**.

An exemplary layer with $M = 2$ and $D = 3$:

$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$



$$Wx = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

$$w = (w_1 \cdots)$$

$$w^T x$$

If we forget the historical interpretation as *neurons*, we can use a clearer algebraic / tensorial formulation:

$$y = \sigma(w \cdot x + b)$$

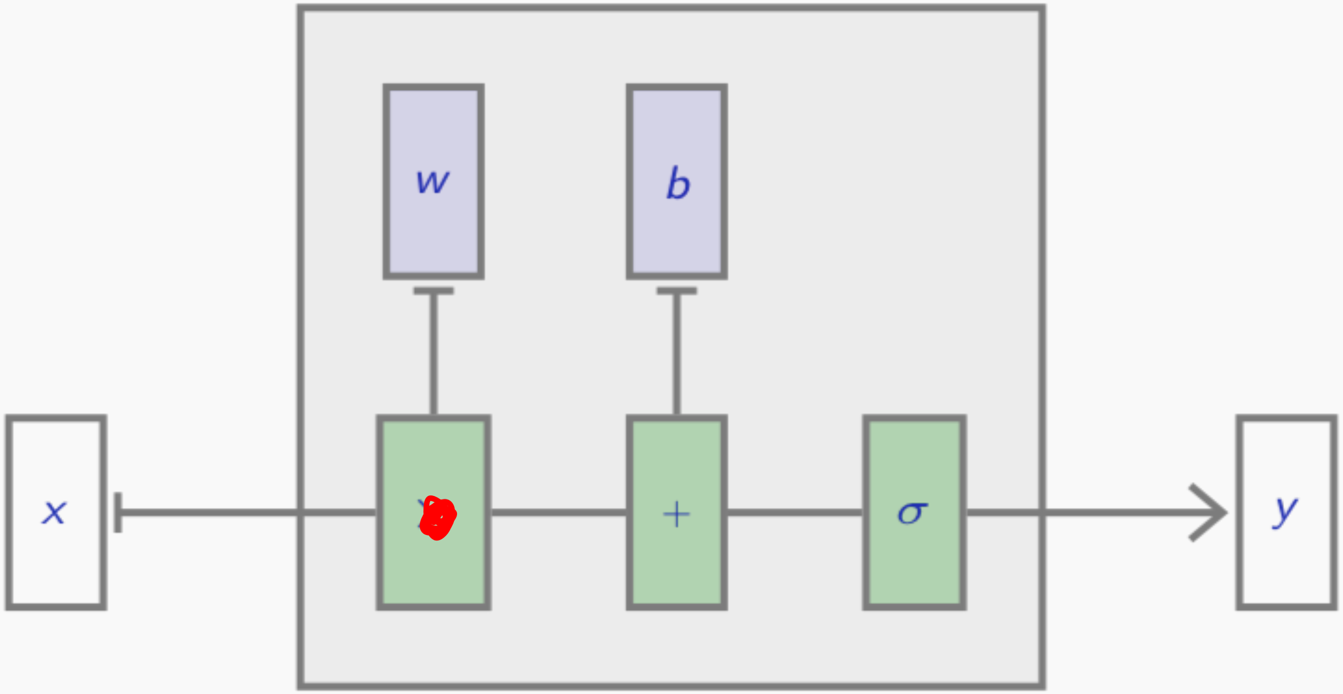
where $x \in \mathbb{R}^D$, $w \in \mathbb{R}^{M \times D}$, $b \in \mathbb{R}^M$, $y \in \mathbb{R}^M$, and σ denotes a component-wise extension of the $\mathbb{R} \to \mathbb{R}$ mapping:

$$\sigma : (y_1, \ldots, y_M) \mapsto (\sigma(y_1), \ldots, \sigma(y_M))$$

$g : \mathbb{N} \rightarrow \mathbb{R}$ $\quad$ $g(5) = 7{,}5$

overload: $g : \mathbb{N} \times \ldots \times \mathbb{N} \longrightarrow \mathbb{R} \times \ldots \times \mathbb{R}$

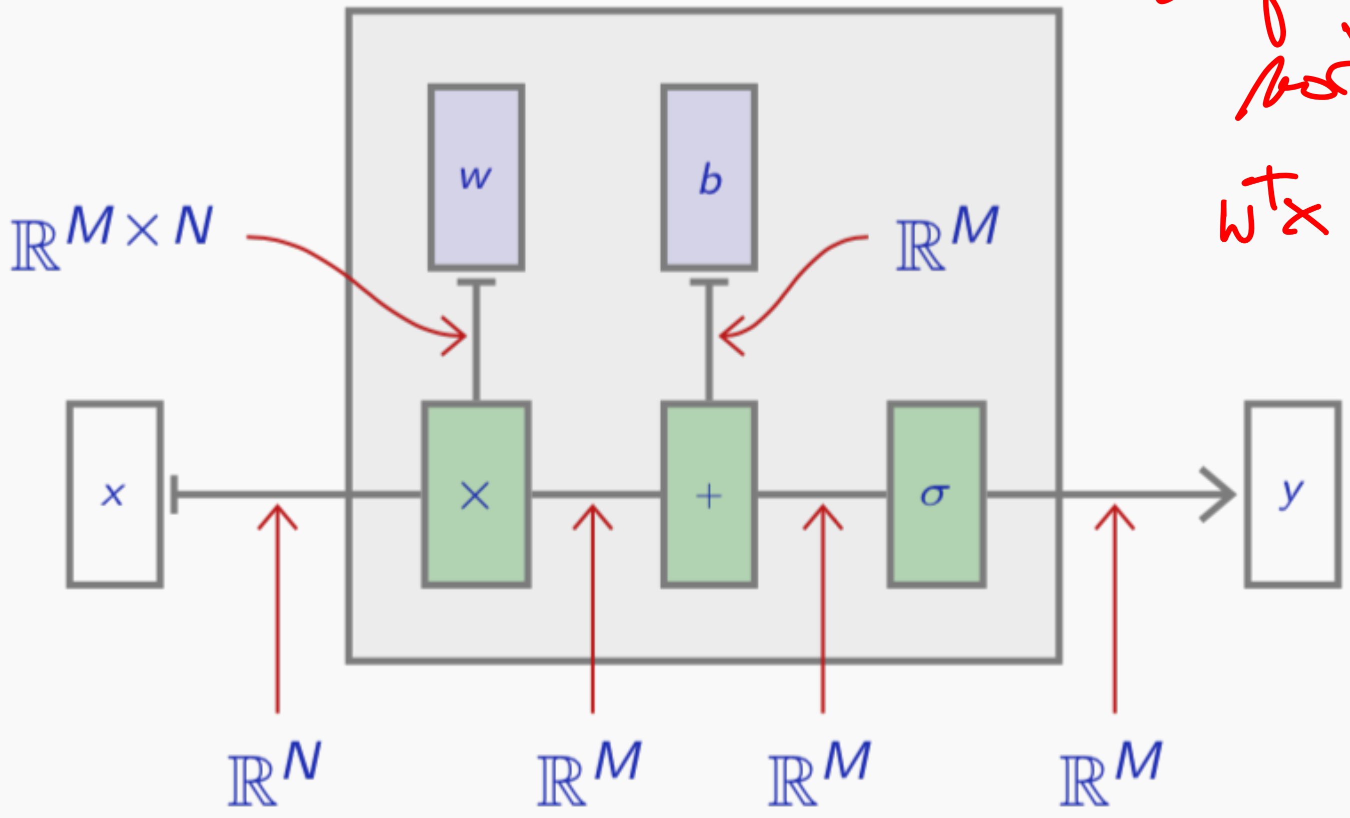
With '$\times$' for the matrix-vector product, the *tensorial block figure* remains almost identical to that of the single neuron.



With '$\times$' for the matrix-vector product, the *tensorial block figure* remains almost identical to that of the single neuron.

Compare with this notation:

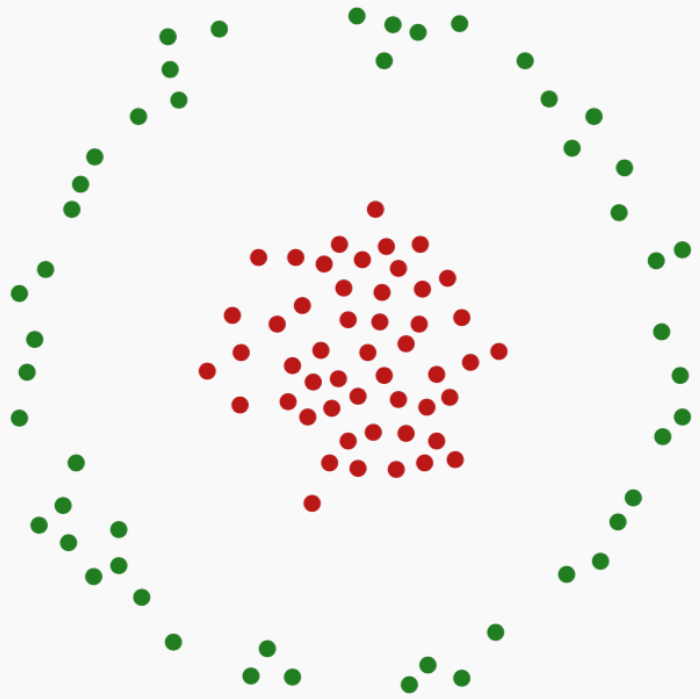
$w^T x \to Wx$

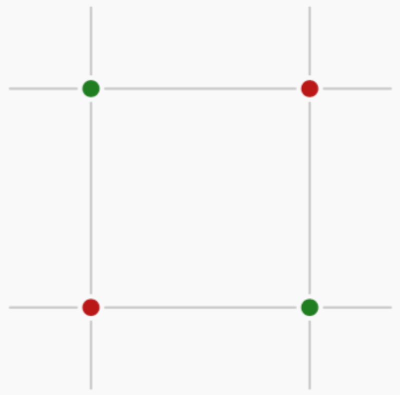


(Note that $N = D$ on the previous slides; N seems to be better here though there is a conflict with the number of training instances)

Limitations of linear predictors and feature design

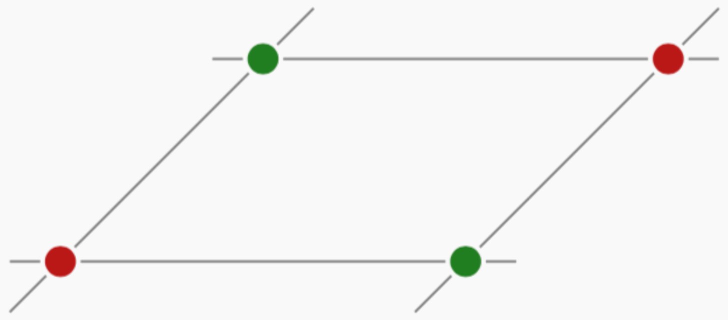
The main weakness of linear predictors is their lack of capacity. For classification, the populations have to be **linearly separable**.





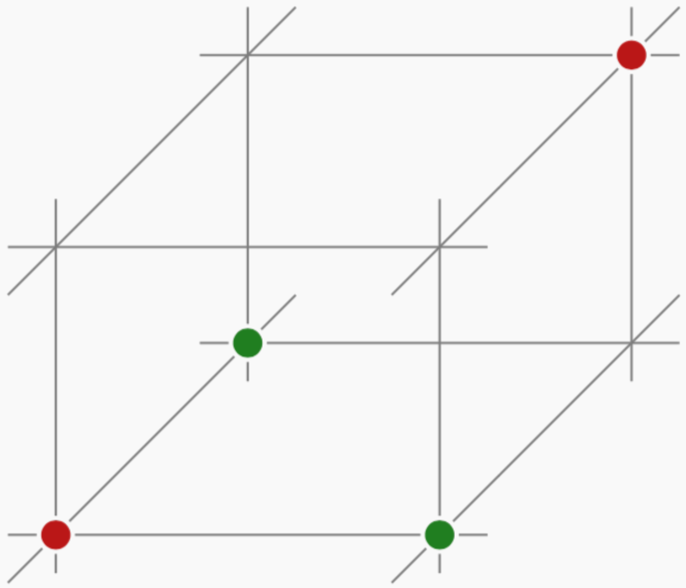
“xor”

The xor example can be solved by pre-processing the data to make the two populations linearly separable:



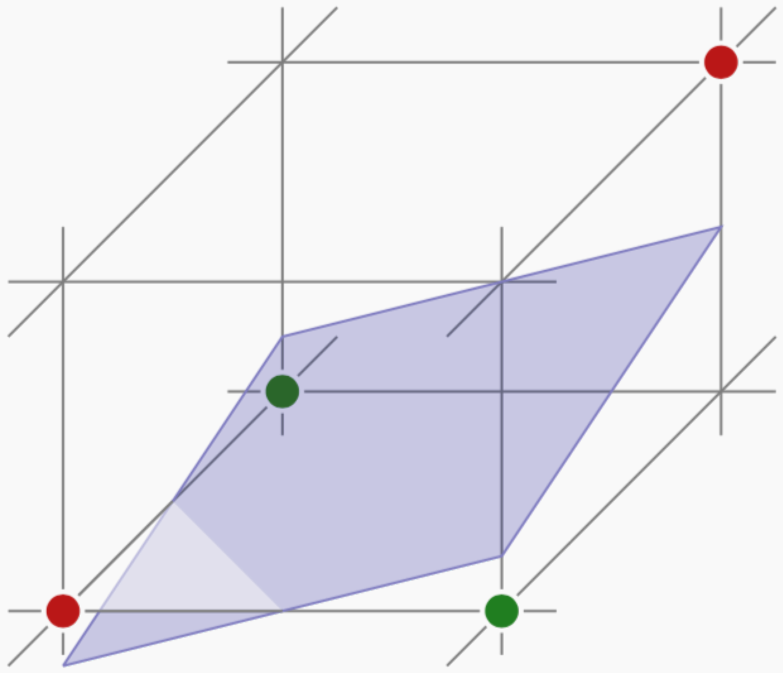
The xor example can be solved by pre-processing the data to make the two populations linearly separable:

$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v)$$



The xor example can be solved by pre-processing the data to make the two populations linearly separable:

$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v)$$



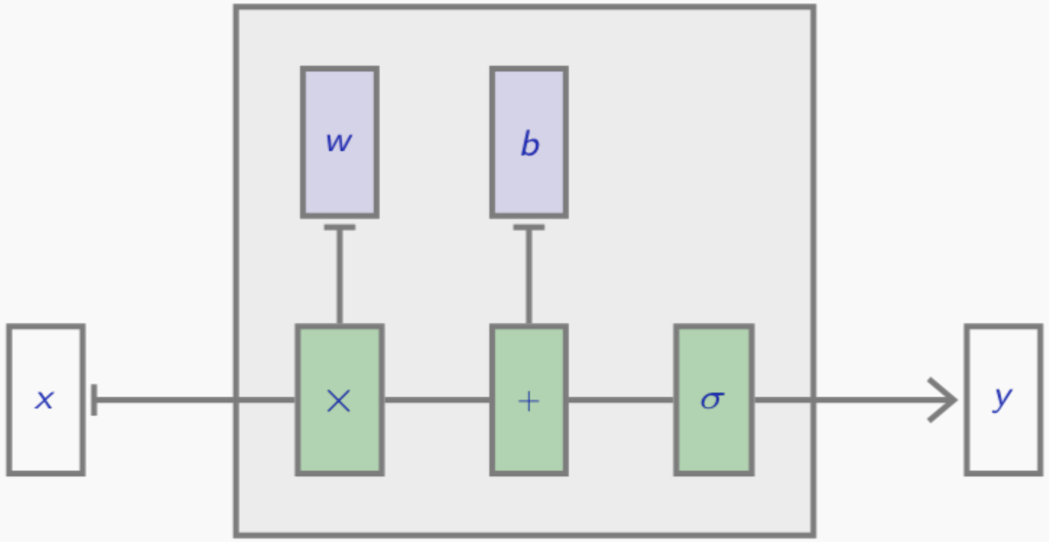
So we can model the xor with

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

Note that the model

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

still fits to the tensorial block figure:



This is similar to the polynomial regression. If we have

$$\Phi : x \mapsto (1, x, x^2, \ldots, x^D)$$

and

$$\alpha = (\alpha_0, \ldots, \alpha_D)$$

then

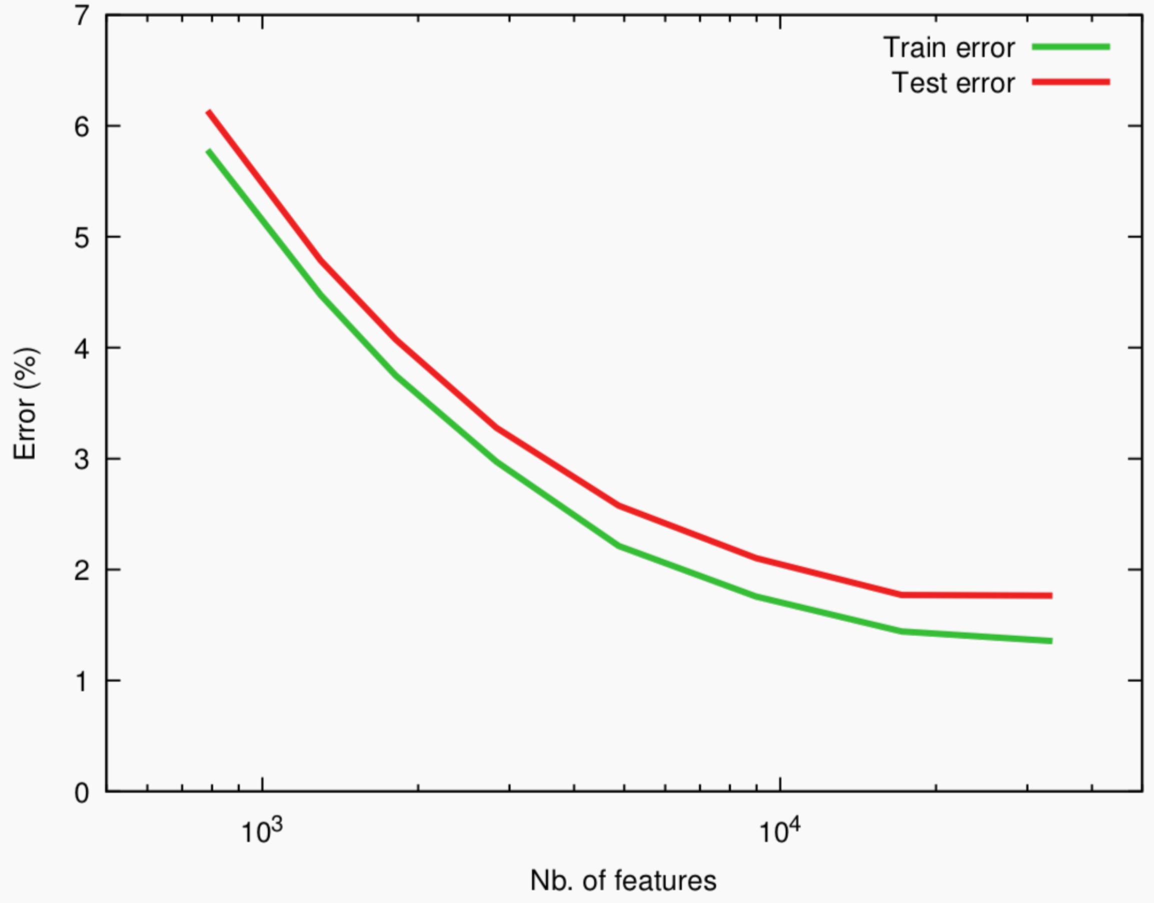
$$\sum_{d=0}^{D} \alpha_d x^d = \alpha \cdot \Phi(x).$$

By increasing D, we can approximate any continuous real function on a compact space (Stone-Weierstrass theorem).

It means that we can make the capacity as high as we want.

We can apply the same to a more realistic binary classification problem: MNIST's '8' vs. the other classes with a perceptron.

The original 28×28 features are supplemented with the products of pairs of features taken at random.



Remember the bias-variance trade-off:

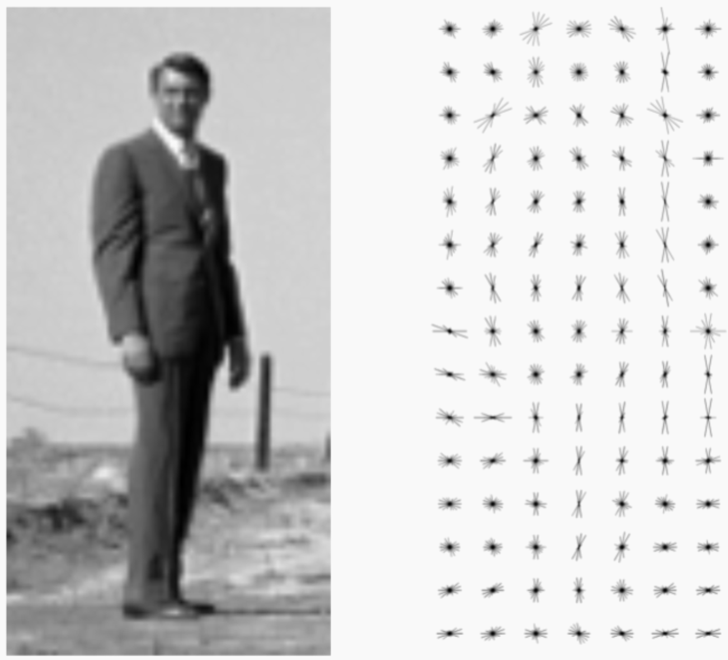
$$\mathbb{E}\left((Y-y)^2\right) - \underbrace{(\mathbb{E}(Y)-y)^2}_{\textit{Bias}} + \underbrace{\mathbb{V}(Y)}_{\textit{Variance}} \; .$$

The right class of models reduces the bias more and increases the variance less. Besides increasing capacity to reduce the bias, *feature design* may also be a way of reducing capacity without hurting the bias, or with improving it.

In particular, good features should be invariant to perturbations of the signal known to keep the value to predict unchanged.

A classical example is the *Histogram of Oriented Gradient* descriptors (HOG), initially designed for person detection.

Roughly: divide the image in 8×8 blocks, compute in each the distribution of edge orientations over 9 bins.



Dalal and Triggs (2005) combined them with an SVM, and Dollár et al. (2009) extended them with other modalities into the *channel features*.

Many methods (perceptron, SVM, k-means, PCA, etc.) only require to compute $\kappa(x, x') = \Phi(x) \cdot \Phi(x')$ for any (x, x').

So one needs to specify κ alone, and may keep Φ undefined.

This is the **kernel trick**, which we will not cover in this course.

Training a model composed of manually engineered features and a parametric model such as logistic regression is now referred to as *shallow learning*.

The signal goes through a single processing trained from data.

References

N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Conference on Computer Vision and Pattern Recognition (CVPR), pages 886–893, 2005.

P. Dollár, Z. Tu, P. Perona, and S. Belongie. Integral channel features. In British Machine Vision Conference, pages 91.1–91.11, 2009.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

F. Rosenblatt. The perceptron–A perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957.