Accelerating first-principles based molecular dynamics simulations with machine learning

Menno Bokdam

University of Twente

m.bokdam@utwente.nl

Winterschool for Theoretical Chemistry and Spectroscopy
Han-sur-Lesse, November 27th, 2024

Outline of my Talks

- Wednesday 17-19h: Introduction to Machine Learning
- Thursday 10-11h: Ab-initio Molecular Dynamics
- Thursday 11-13h: On-the-fly Machine Learned Force Fields

Outline for today

1 Outline

2 Math notation

3 Machine 'Learning'

4 Linear Regression

5 Logistic Regression

6 Regression craftmanship

7 Neural Networks

8 Optional: build your own NN classifier

Reference materials

- A Machine Learning book that I recommend is: *Pattern Recognition and Machine Learning*, by Christopher M. Bishop. (Free PDF version, sponsored by Microsoft)

Mathematics notation (1/4)

- Scalars are written in lowercase (η) or uppercase (L)
- *Column* vectors are written in bold lowercase ($\mathbf{b}$)
- *Row* vectors are written in bold lowercase transpose ($\mathbf{b}^{\mathrm{T}}$)
- Elements of vectors are in lowercase with subscript (b_i)

Examples:

$$\mathbf{a} = \begin{bmatrix} 2 \\ 2.1 \\ 8 \\ 5 \end{bmatrix}, \qquad \mathbf{b}^{\mathrm{T}} = \begin{bmatrix} b_0 & b_1 & b_2 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}.$$

$\dim(\mathbf{a}) = 4 \times 1$

Mathematics notation (2/4)

- The norm or length of a vector is written as $\|\mathbf{b}\|$ and is most of the time the $\ell_2(\mathbb{R})$-norm, ie. $\|\mathbf{b}\|_2 = \sqrt{\sum_{i=0}^{N} |b_i|^2}$.
- The inner product of two vectors: $\langle \mathbf{a}|\mathbf{b}\rangle = \mathbf{a}^{\mathrm{T}} \cdot \mathbf{b} = \sum_{i=0}^{N} a_i b_i$ is most of the time the standard scalar product
- Element-wise product ('Hadamard') $\mathbf{a} \circ \mathbf{b} = \mathbf{c}$

Examples:

$$\mathbf{a} = \begin{bmatrix} 2 \\ 2.1 \\ 8 \\ 5 \end{bmatrix}, \|\mathbf{a}\| = \sqrt{2^2 + 2.1^2 + 8^2 + 5^2} \approx 9.86965045,$$

$$\langle \mathbf{c}|\mathbf{b}\rangle = \mathbf{c}^{\mathrm{T}} \cdot \mathbf{b} = \begin{bmatrix} c_0 & c_1 & c_2 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = c_0 b_0 + c_1 b_1 + c_2 b_2 = \|\mathbf{c} \circ \mathbf{b}\|^2$$

Mathematics notation (3/4)

- Matrices are written in bold uppercase ($\mathbf{W}$)
- Elements of matrices are in uppercase with subscript (W_{ij}, with i the row and j the column) or as $\mathbf{W}[i, j]$
- $\mathbf{W}[i, :]$ denotes the i'th *row* of the matrix $\mathbf{W}$
- $\mathbf{W}[:, j]$ denotes the j'th *column* of the matrix $\mathbf{W}$

Examples:

$$\mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix}, \mathbf{W}[2, :] = \begin{bmatrix} W_{2,1} & W_{2,2} \end{bmatrix}, \mathbf{W}[:, 2] = \begin{bmatrix} W_{1,2} \\ W_{2,2} \end{bmatrix}$$

$$\mathbf{Ra} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} Aa_1 + Ba_2 \\ Ca_1 + Da_2 \end{bmatrix}$$

Mathematics notation (4/4)

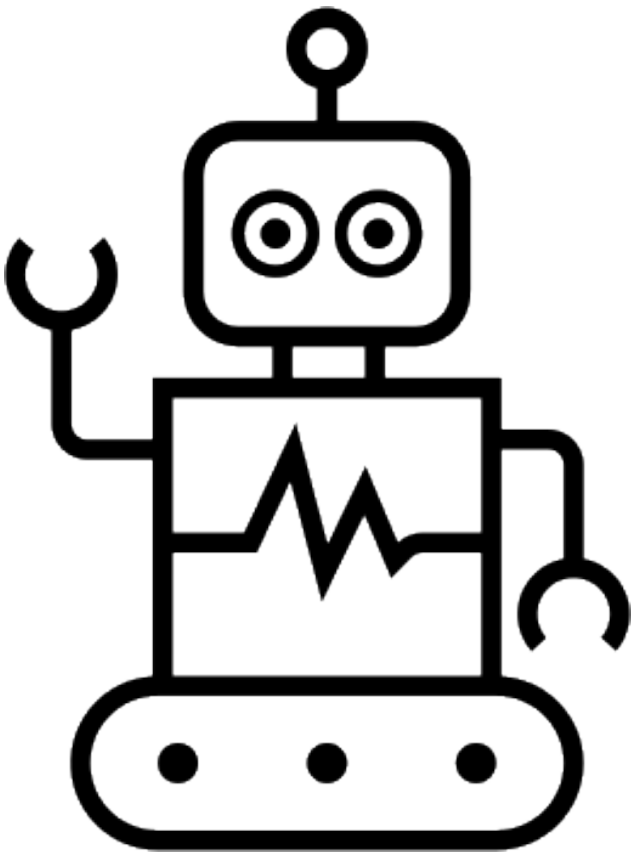
- Note that: $\mathbf{b}^{\mathrm{T}} \cdot \mathbf{b}$ =scalar and $\mathbf{b} \cdot \mathbf{b}^{\mathrm{T}}$ =matrix.

Examples:

$$\mathbf{a} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \mathbf{b} = \begin{bmatrix} d \\ e \\ f \end{bmatrix}, \mathbf{a}^{\mathrm{T}} \cdot \mathbf{b} = \begin{bmatrix} a & b & c \end{bmatrix} \cdot \begin{bmatrix} d \\ e \\ f \end{bmatrix} = ad + be + cd,$$

$$\mathbf{a} \cdot \mathbf{b}^{\mathrm{T}} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \cdot \begin{bmatrix} d & e & f \end{bmatrix} = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

OK, Lets start with ML!



Machine Learning

'Learning' can happen in (at least) four ways

- Supervised learning;
- Unsupervised learning;
- Reinforcement learning;
- Deep learning.

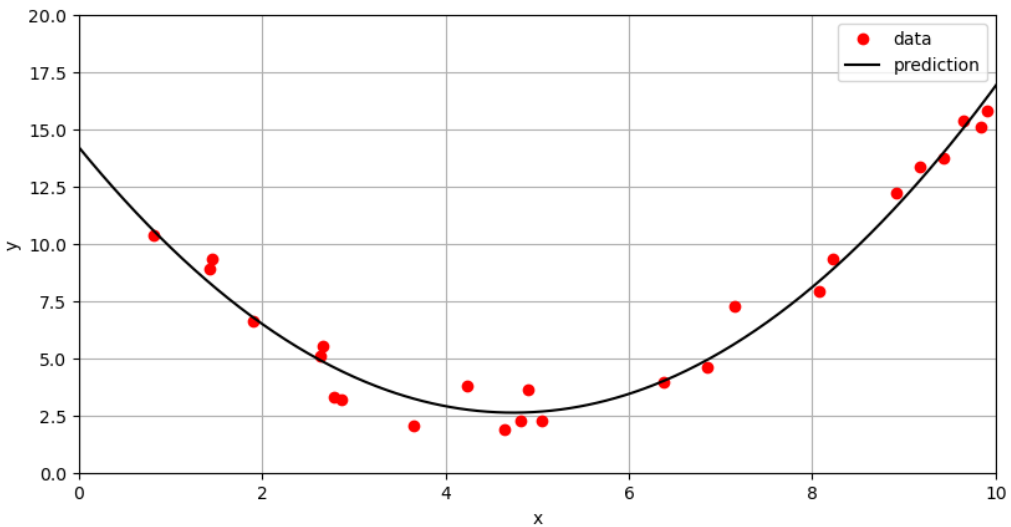
Supervised learning.

You are given data $\mathbf{x}_i$, $i = 1, \ldots, M$ with a label (binary: $0, 1$, multiclass: $0, 1, 2, \ldots$)

- You choose a method.
- Training dataset: The sample of data used to fit the model/using the method.
- Validation dataset: Is your chosen model/method 'correct'?
- Test set: How well does the model predict the class of that data?

ML jargon

Suppose we receive a set of $M = 25$ measurements of some observable y depending on the variable x, then the 'ML expert' would say: You have a **dataset** containing 25 **samples** (or **instances**). We are going to *train*, by **supervised learning**, a continuous model that maps the **feature** x to its **label** y in the 'best' way we can, but with the lowest possible complexity of the model.



Ok, isn't that just curve fitting?

Yes, and a good starting place to study ML methods.

Lets define some linear Algebra:

$$\mathbf{y} = \begin{bmatrix} y^{(0)} \\ y^{(1)} \\ .. \\ y^{(m)} \\ .. \\ y^{(M-1)} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x^{(0)} \\ x^{(1)} \\ .. \\ x^{(m)} \\ .. \\ x^{(M-1)} \end{bmatrix} \tag{1}$$

We can then build the model:

$$\mathbf{y} = a\mathbf{x} \tag{2}$$

However, considering our data this doesn't suffice.

A 2nd order polynomial would be better: (Note, Hadamard (element-wise) product $\mathbf{x} \circ \mathbf{x}$)

$$\mathbf{y} = \theta_2 \mathbf{x} \circ \mathbf{x} + \theta_1 \mathbf{x} + \theta_0 \tag{3}$$

To fit this we have to add $N = 3$ (derived) *features* to our **feature vector** $\mathbf{x}^{(m)}$. This then gives the **design matrix**:

$$\mathbf{X} = \begin{bmatrix} (x^{(0)})^0 & (x^{(0)})^1 & (x^{(0)})^2 \\ (x^{(1)})^0 & (x^{(1)})^1 & (x^{(1)})^2 \\ .. & & \\ (x^{(m)})^0 & (x^{(m)})^1 & (x^{(m)})^2 \\ .. & & \\ (x^{(M-1)})^0 & (x^{(M-1)})^1 & (x^{(M-1)})^2 \end{bmatrix} = \begin{bmatrix} (\mathbf{x}^{(0)})^{\mathrm{T}} \\ (\mathbf{x}^{(1)})^{\mathrm{T}} \\ .. \\ (\mathbf{x}^{(m)})^{\mathrm{T}} \\ .. \\ (\mathbf{x}^{(M-1)})^{\mathrm{T}} \end{bmatrix} \tag{4}$$

(This particular matrix is also known as a *vanderMonde matrix*.)

We can now build a linear model:

$$\mathbf{y} = \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X}) = \mathbf{X}\boldsymbol{\theta}, \tag{5}$$

with $\mathbf{h}_{\boldsymbol{\theta}}$ the *hypothesis function* and *weight vector* $\boldsymbol{\theta}$.

The loss function

We now need to find those weights $\boldsymbol{\theta}$ such that the *error* is small when averaged over all M:

$$\boldsymbol{\epsilon} = \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X}) - \mathbf{y} \qquad \epsilon = \frac{1}{M}\sum_{m=0}^{M-1} \epsilon_m \tag{6}$$

In practice, the **loss function** is used:

$$\begin{aligned} L &= \frac{1}{M}\frac{1}{2}\left\|\boldsymbol{\epsilon}\right\|^2 \\ L &= \frac{1}{M}\frac{1}{2}\left\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right\|^2 \end{aligned} \tag{7}$$

You might recognize the "least-squares-method". Minimizing the loss wrt. to the weights will enable us to optimize the model.
l_2 norm: $\|\mathbf{x}\| = \sqrt{x_0^2 + x_1^2 \ldots}$

Minimizing the loss function

At minima of the loss we have $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = 0$.

Theorem

The vector $\mathbf{x} \in \mathbb{R}^n$ *is a solution of the linear optimisation problem* $\|\mathbf{b} - \mathbf{A}\mathbf{x}\| = min$, *if and only if, it statisfies the normal equations:*

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{8}$$

$$\mathbf{X}\boldsymbol{\theta} = \mathbf{y} \tag{9}$$

Minimizing the loss function

At minima of the loss we have $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = 0$.

Theorem

The vector $\mathbf{x} \in \mathbb{R}^n$ *is a solution of the linear optimisation problem* $\|\mathbf{b} - \mathbf{A}\mathbf{x}\| = min$, *if and only if, it statisfies the normal equations:*

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{10}$$

$$\begin{aligned} \mathbf{X}\boldsymbol{\theta} &= \mathbf{y} \\ \mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{\theta} &= \mathbf{X}^{\mathrm{T}}\mathbf{y} \end{aligned} \tag{11}$$

Minimizing the loss function

At minima of the loss we have $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = 0$.

Theorem

The vector $\mathbf{x} \in \mathbb{R}^n$ *is a solution of the linear optimisation problem* $\|\mathbf{b} - \mathbf{A}\mathbf{x}\| = min$, *if and only if, it statisfies the normal equations:*

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{12}$$

$$\begin{aligned} \mathbf{X}\boldsymbol{\theta} &= \mathbf{y} \\ \mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{\theta} &= \mathbf{X}^{\mathrm{T}}\mathbf{y} \\ (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{\theta} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \end{aligned} \tag{13}$$

Minimizing the loss function

At minima of the loss we have $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = 0$.

Theorem

The vector $\mathbf{x} \in \mathbb{R}^n$ *is a solution of the linear optimisation problem* $\|\mathbf{b} - \mathbf{A}\mathbf{x}\| = min$, *if and only if, it statisfies the normal equations:*

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{14}$$

$$\begin{aligned} \mathbf{X}\boldsymbol{\theta} &= \mathbf{y} \\ \mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{\theta} &= \mathbf{X}^{\mathrm{T}}\mathbf{y} \\ (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{\theta} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \\ \boldsymbol{\theta} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \end{aligned} \tag{15}$$

Complications with $\mathbf{X}$: overdetermined (ie. not square) and can be huge!
Moore-Penrose inverse: $\mathbf{X}^{+} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}$

Minimizing by gradient descent

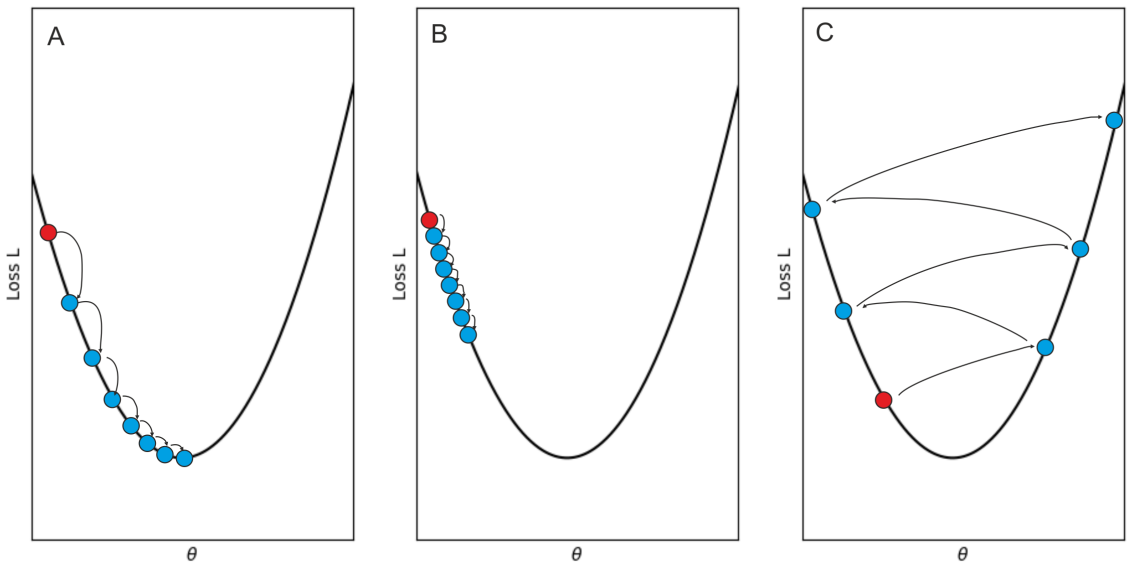
1. Guess initial $\boldsymbol{\theta}$
2. Optimize: $\boldsymbol{\theta}^{new} = \boldsymbol{\theta}^{old} - \eta \frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \boldsymbol{\theta}^{old} - \eta \nabla L$
3. Compute new loss
4. If $|L^{old} - L^{new}| \leq$ some threshold, then stop, else go to 2.

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial \theta_0} \\ \frac{\partial L}{\partial \theta_1} \\ .. \\ \frac{\partial L}{\partial \theta_{N-1}} \end{bmatrix}, \quad \frac{\partial L}{\partial \theta_0} = \lim_{\Delta \to 0} \frac{L(\boldsymbol{\theta} + \boldsymbol{\Delta}_0) - L(\boldsymbol{\theta})}{\Delta}, \quad \boldsymbol{\Delta}_0 = \begin{bmatrix} \Delta \\ 0 \\ .. \\ 0 \end{bmatrix} \tag{16}$$

or more conveniently:

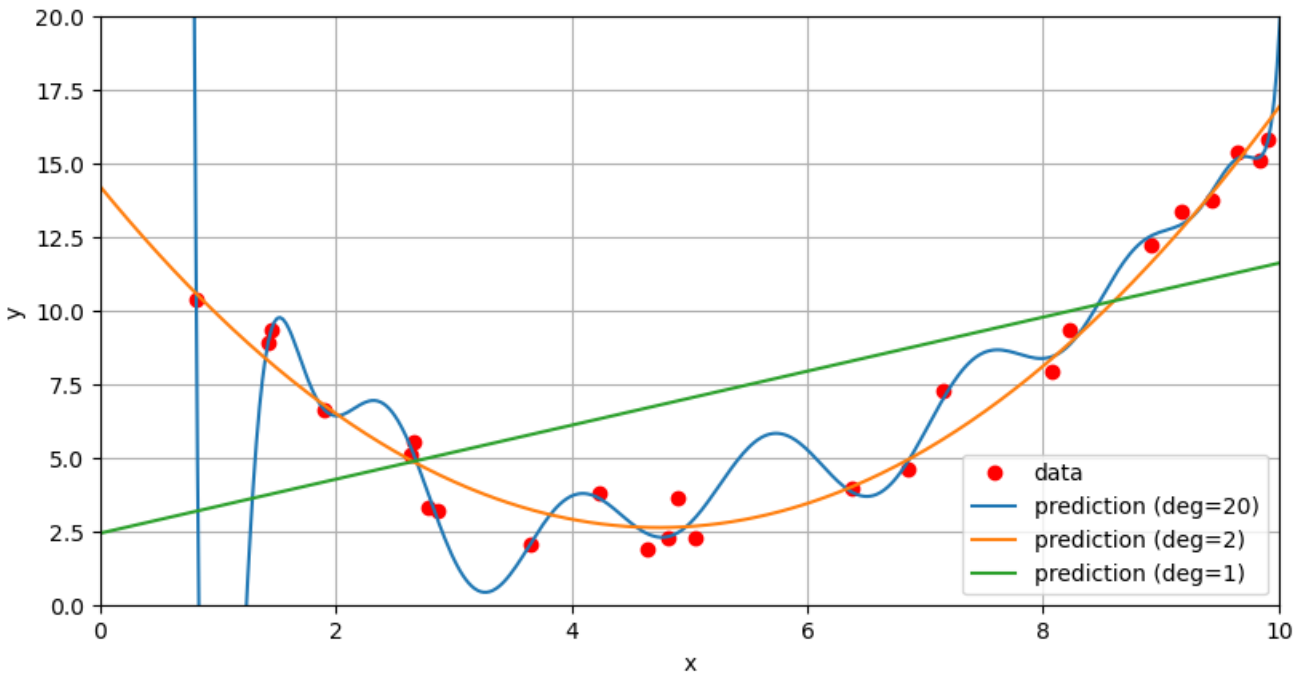
$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \frac{1}{M} \mathbf{X}^{\mathrm{T}} (\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) \tag{17}$$

Minimizing by gradient descent



Setting the '*friction coefficient*' makes a difference:
A. η perfect.
B. η too small.
C. η too large.

Learned/fitted model



Be careful when expanding the complexity of your ML model. As shown here the model of deg=20 shows serious signs of **overfitting**. You can recognize this problem when you split your dataset in a **training** and a **validation** set and compare the errors.

The regularized loss function

To reduce overfitting, add an additional constrain that suppresses large weight values θ_i:

$$L = \frac{1}{M}\frac{1}{2}\left\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right\|^2 + \frac{\lambda}{2}\left\|\boldsymbol{\theta}\right\|^2 \tag{18}$$

The Regularization parameter (λ) has to be set carefully.

The regularized loss function

To reduce overfitting, add an additional constrain that suppresses large weight values θ_i:

$$L = \frac{1}{M}\frac{1}{2}\left\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right\|^2 + \frac{\lambda}{2}\left\|\boldsymbol{\theta}\right\|^2 \tag{19}$$

To find the optimal weights we proceed as follows:

$$\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} &= 0 \\
\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}\left(\frac{1}{M}\frac{1}{2}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^{\mathrm{T}}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) + \frac{\lambda}{2}\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{\theta}\right) &= 0 \\
\frac{1}{M}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) + \lambda\boldsymbol{\theta} &= 0 \\
\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y} + \lambda M\boldsymbol{\theta} &= 0 \\
(\mathbf{X}^T\mathbf{X} + \lambda M\mathbf{I})\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y} &= 0 \\
\boldsymbol{\theta} &= (\mathbf{X}^T\mathbf{X} + \lambda M\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}
\end{aligned} \tag{20}$$

The regularized loss function

To reduce overfitting, add an additional constrain that suppresses large weight values θ_i:

$$L = \frac{1}{M}\frac{1}{2}\left\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right\|^2 + \frac{\lambda}{2}\left\|\boldsymbol{\theta}\right\|^2 \tag{21}$$

To find the optimal weights we proceed as follows:

$$\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X} + \lambda M\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{22}$$

Note that if the mean$(\mathbf{y}) = \mu \neq 0$, then the loss cannot become 0. The minimum loss will then be $|\theta_0|$, which is only reached when all $\theta_{i\geq 1} = 0$. Therefore, it is advisable to preform feature scaling in advance, simply use $\mathbf{y}' = \mathbf{y} - \frac{1}{M}\sum_{m=1}^{M} y^{(m)}$ in Equation (46). An alternative would be to not use θ_0 when computing $\|\boldsymbol{\theta}\|$.

Algorithms for finding the least-squares solution

$$\mathbf{X}\boldsymbol{\theta} = \mathbf{y}, \quad \mathbf{X} \in \mathbb{C}^{m \times n} \tag{23}$$

- Gradient decent (cheap, no guaranteed minimum)
- LU decomposition, $\mathcal{O}(\frac{2}{3}n^3)$
- Cholesky decomposition, $\mathcal{O}(\frac{1}{3}n^3)$
- QR decomposition, $\mathcal{O}(2mn^2 - \frac{2}{3}n^3)$
- Singular value decomposition, $\mathcal{O}(2mn^2 + 11n^3)$
- ...

Computation costs: Nick Higham, "Functions of Matrices Theory and Computation"

'Exact' solution: QR decomposition

If the number of samples is small, you can decompose the feature vector:

$$\mathbf{X} = \mathbf{QR}. \tag{24}$$

These matrices have nice properties, $\mathbf{Q} = \mathbf{Q}^{\mathrm{T}}$, $\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{Q}\mathbf{Q}^{\mathrm{T}} = \mathbf{I}$, $\mathbf{Q}^2 = \mathbf{I}$, and $\mathbf{R}$ is an upper-triangular matrix. Hereafter you can solve the normal equation up to machine precision.

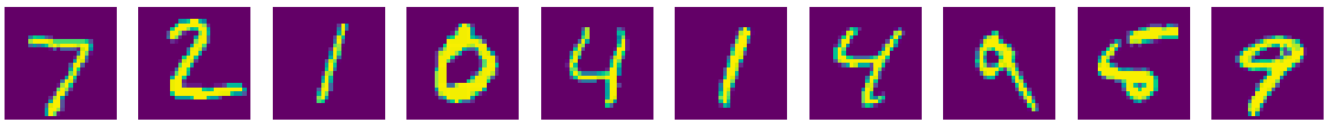
$$\begin{aligned}
\mathbf{X}\boldsymbol{\theta} &= \mathbf{y} \\
\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{\theta} &= \mathbf{X}^{\mathrm{T}}\mathbf{y} \\
(\mathbf{QR})^{\mathrm{T}}\mathbf{QR}\boldsymbol{\theta} &= (\mathbf{QR})^{\mathrm{T}}\mathbf{y} \\
\mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{QR}\boldsymbol{\theta} &= \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{y} \quad , \quad \mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{I} \\
\mathbf{R}^{\mathrm{T}}\mathbf{R}\boldsymbol{\theta} &= \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{y} \\
\mathbf{R}\boldsymbol{\theta} &= \mathbf{Q}^{\mathrm{T}}\mathbf{y} = \tilde{\mathbf{y}}
\end{aligned} \tag{25}$$

In the last line you can solve for $\boldsymbol{\theta}$ by backwards-substitution.

Classification

In classification the task is to predict *classes*.

Multi-class classification:



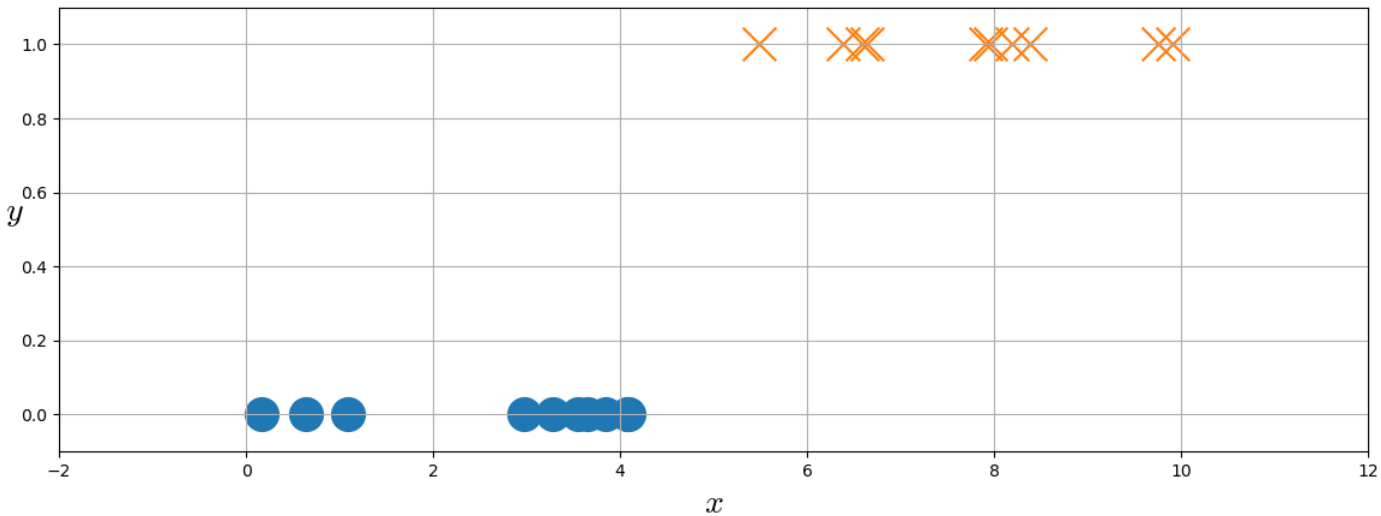
Labels: $y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Binary classification:

- Is an email spam or not?
- Is a patient suffering from a condition or not?

Labels: $y \in \{0, 1\}$

Hypothesis function



Linear regression: $\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{x}^T\boldsymbol{\theta}$

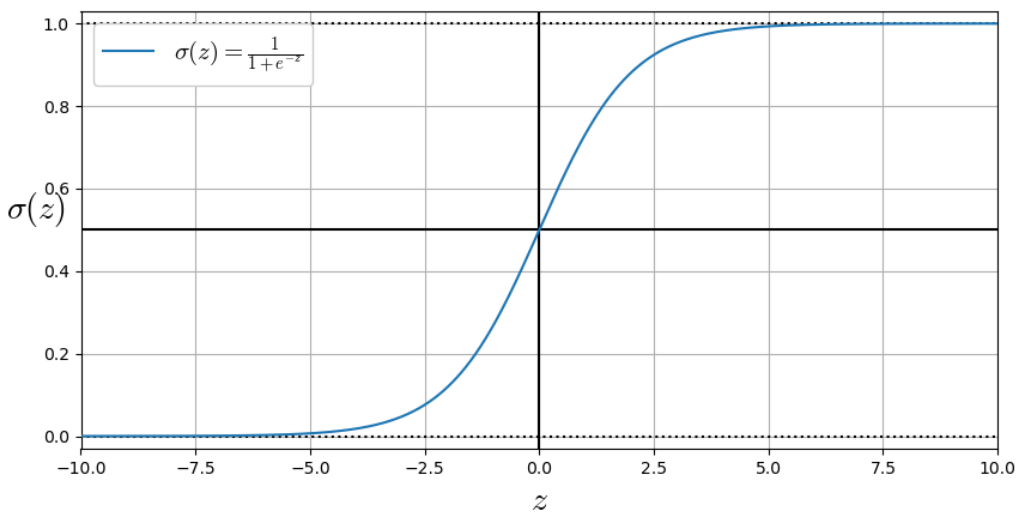
Logistic regression: $\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = f(\mathbf{x}^T\boldsymbol{\theta})$

function f maps $\mathbf{x}^T\boldsymbol{\theta}$ on the interval $[0, 1]$

Hypothesis function

Logistic regression uses the *sigmoid* function (or Logistic function):

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{26}$$



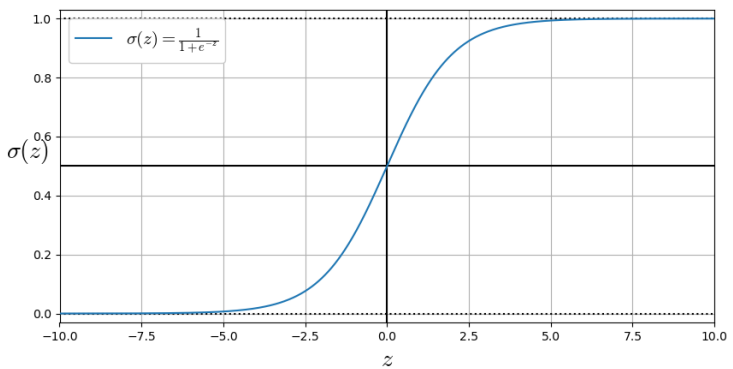
Hypothesis function

Logistic regression uses the *sigmoid* function (or Logistic function):

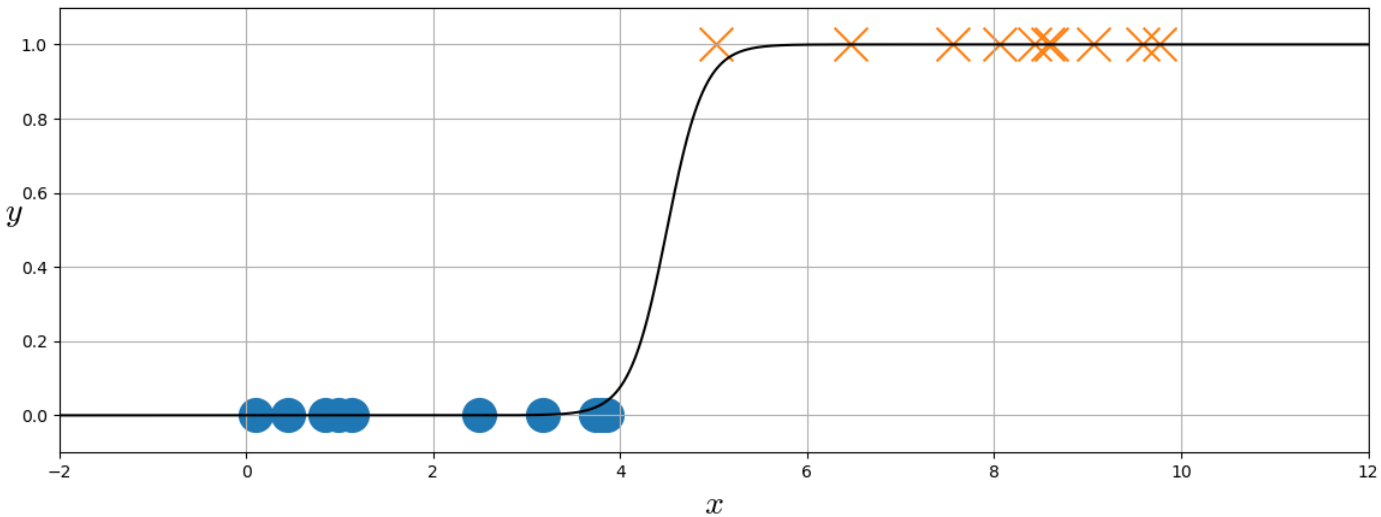
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Its derivative can be calculated using the *sigmoid* function:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z)) \tag{27}$$



Hypothesis function



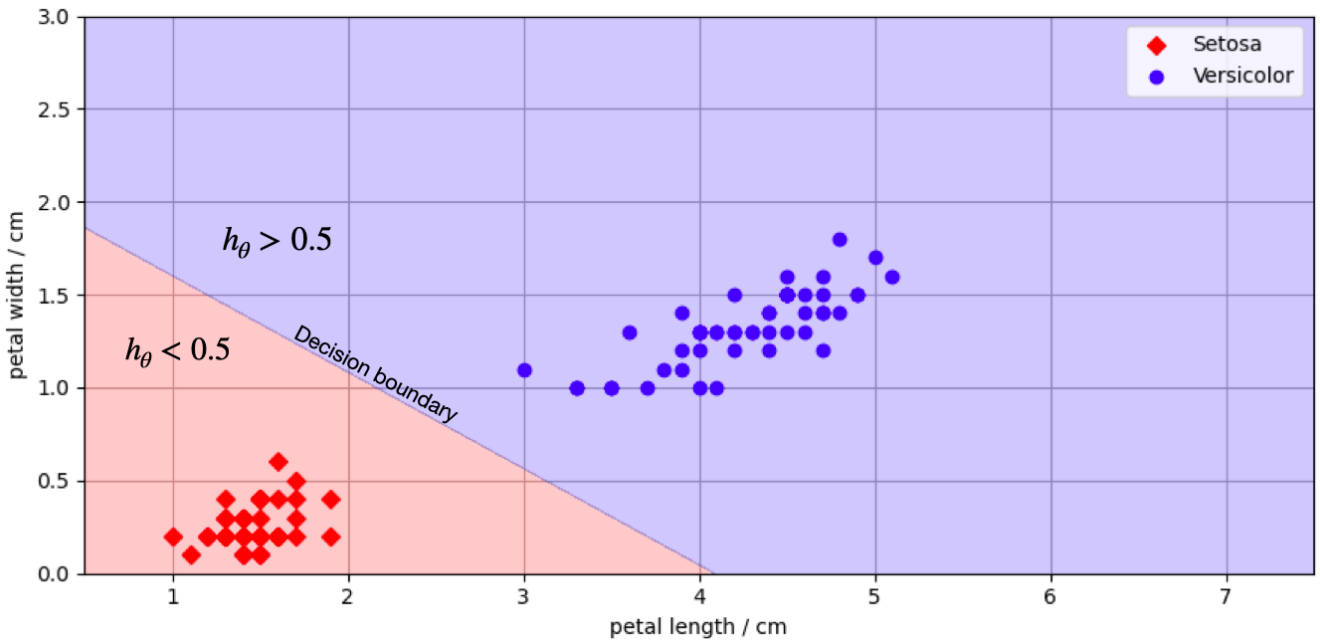
$h_\theta(z) = \frac{1}{1+e^{-z}}$, with $z = \mathbf{x}^T\boldsymbol{\theta}$

Predict "1" if $h_\theta(\mathbf{x}) \geq 0.5$ or:
$\mathbf{x}^T\boldsymbol{\theta} \geq 0$
Predict "0" if $h_\theta(\mathbf{x}) < 0.5$ or:
$\mathbf{x}^T\boldsymbol{\theta} < 0$

Decision boundary

The decision boundary: $\mathbf{x}^T\boldsymbol{\theta} = 0$

Assume 2 features plus the bias: $\mathbf{x}^T\boldsymbol{\theta} = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 = 0$



Multiple samples

As in lin. reg, we represent the whole dataset as follows:

$$\mathbf{y} = \begin{bmatrix} y^{(0)} \\ y^{(1)} \\ .. \\ y^{(m)} \\ .. \\ y^{(M-1)} \end{bmatrix}, \mathbf{X} = \begin{bmatrix} \left(\mathbf{x}^{(0)}\right)^T \\ \left(\mathbf{x}^{(1)}\right)^T \\ .. \\ \left(\mathbf{x}^{(m)}\right)^T \\ .. \\ \left(\mathbf{x}^{(M-1)}\right)^T \end{bmatrix}, \tag{28}$$

so in the matrix $\mathbf{X}$ each row contains the features of one sample from the dataset, and in the vector $\mathbf{y}$ contains the corresponding label.

Linear versus Logistic regression model

Linear regression:

$$h_{\boldsymbol{\theta}}(\mathbf{X}) = \mathbf{X}\theta \tag{29}$$

Logistic regression:

$$h_{\boldsymbol{\theta}}(\mathbf{X}) = \frac{1}{1 + e^{-\mathbf{X}\theta}} \tag{30}$$

The cross-entropy loss function (a.k.a. log loss)

The entropy ($S = -k_B \sum_i p_i \ln p_i$) function for a binary classification:

$$L_{\mathrm{CE}} = y^{(m)} \ln p_1^{(m)} + y^{(m)} \ln p_0^{(m)}$$

The cross-entropy loss function (a.k.a. log loss)

The entropy ($S = -k_B \sum_i p_i \ln p_i$) function for a binary classification:

$$L_{\mathrm{CE}} = y^{(m)} \ln p_1^{(m)} + y^{(m)} \ln p_0^{(m)}$$

$$p_1 + p_0 = 1 \implies$$

$$L_{\mathrm{CE}} = y^{(m)} \ln \left(p^{(m)}\right) + \left(1 - y^{(m)}\right) \ln \left(1 - p^{(m)}\right)$$

The cross-entropy loss function (a.k.a. log loss)

The entropy ($S = -k_B \sum_i p_i \ln p_i$) function for a binary classification:

$$L_{\mathrm{CE}} = y^{(m)} \ln p_1^{(m)} + y^{(m)} \ln p_0^{(m)}$$

$$p_1 + p_0 = 1 \implies$$

$$L_{\mathrm{CE}} = y^{(m)} \ln \left(p^{(m)}\right) + \left(1 - y^{(m)}\right) \ln \left(1 - p^{(m)}\right)$$

$$p_1^{(m)} = P(y_m = 1 | \mathbf{x}_m) = h_\theta(\mathbf{x}_m) = \frac{1}{1 + e^{-\mathbf{x}_m^{\mathrm{T}} \theta}} \tag{31}$$

The cross-entropy loss function (a.k.a. log loss)

The entropy ($S = -k_B \sum_i p_i \ln p_i$) function for a binary classification:

$$L_{\mathrm{CE}} = y^{(m)} \ln p_1^{(m)} + y^{(m)} \ln p_0^{(m)}$$

$$p_1 + p_0 = 1 \implies$$

$$L_{\mathrm{CE}} = y^{(m)} \ln \left(p^{(m)}\right) + \left(1 - y^{(m)}\right) \ln \left(1 - p^{(m)}\right)$$

$$L_{\mathrm{CE}} = -\frac{1}{M} \sum_{m=0}^{M-1} y^{(m)} \ln \left[h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)})\right] + \left(1 - y^{(m)}\right) \ln \left[1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)})\right] \tag{32}$$

The cross-entropy loss function (a.k.a. log loss)

$$L_{\mathrm{CE}} = -\frac{1}{M}\sum_{m=0}^{M-1} y^{(m)} \ln\left[h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)})\right] + \left(1 - y^{(m)}\right) \ln\left[1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)})\right]$$

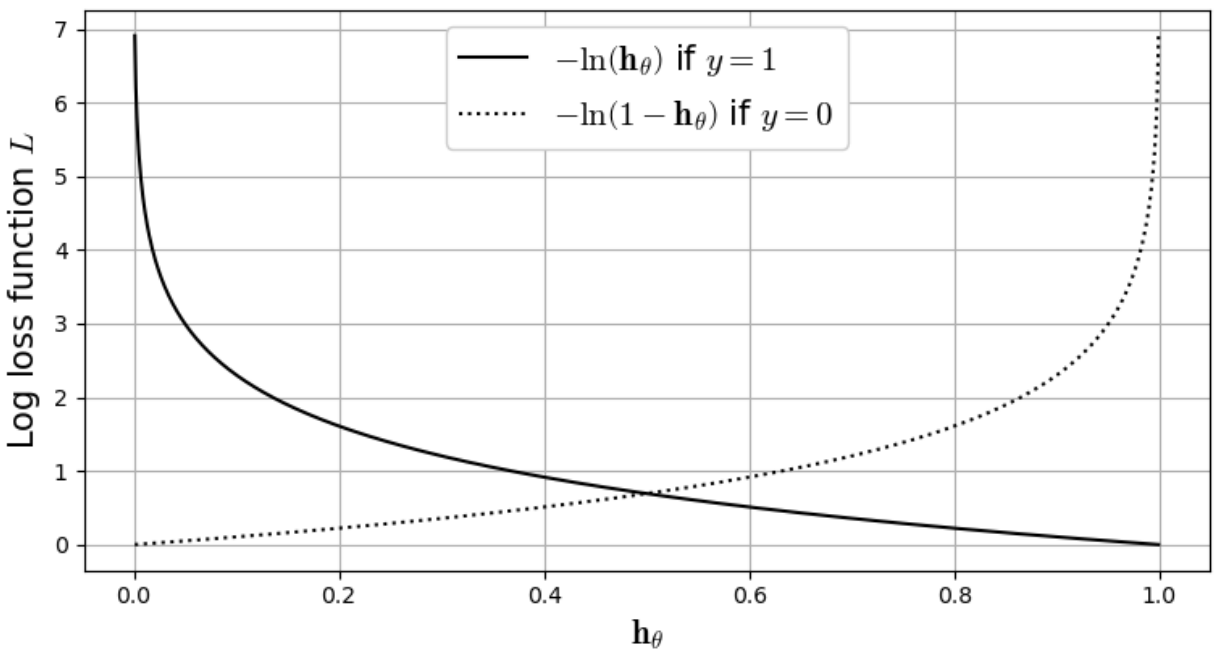
Extending the above equation of the loss to the full data set in matrix notation we have:

$$L_{\mathrm{CE}} = -\frac{1}{M}\left(\mathbf{y}^{\mathrm{T}} \ln\left[\sigma\left(\mathbf{X}\boldsymbol{\theta}\right)\right] + \left(\mathbf{1} - \mathbf{y}\right)^{\mathrm{T}} \ln\left[\mathbf{1} - \sigma\left(\mathbf{X}\boldsymbol{\theta}\right)\right]\right) \qquad (33)$$

Training of the logistic regression model

The loss function:

$$L = -\frac{1}{M}\sum_{m=0}^{M-1} y^{(m)} \ln\left[h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)})\right] + \left(1 - y^{(m)}\right) \ln\left[1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)})\right]$$



Training of the logistic regression model

Use gradient decent to optimise the model. The required derivatives are computed by:

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \frac{1}{M}\mathbf{X}^{\mathrm{T}}\left(\sigma\left(\mathbf{X}\boldsymbol{\theta}\right) - \mathbf{y}\right) \tag{34}$$

, which is similar ($\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \frac{1}{M}\mathbf{X}^{T}\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right)$) to the derivatives we derived for linear regression, but note the important distinction.

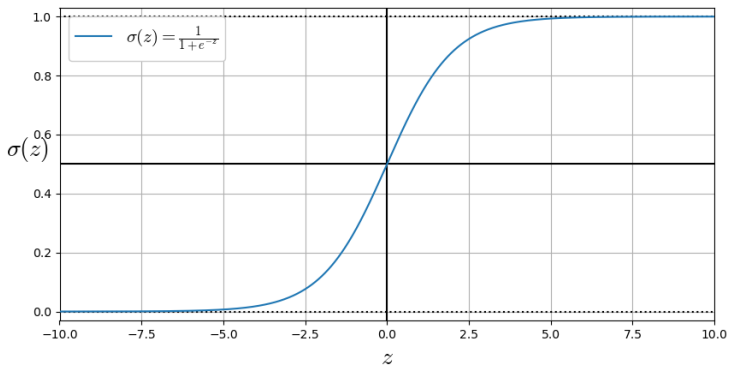
Because of the non-linearity of the logistic model, a 'best' θ vector cannot be computed by a matrix inversion.

Technical improvements (craftsmanship)

Some things you should (always) do:

- Feature scaling
- Polynomial features
- Kernels
- Reguralisation

Feature scaling



$\sigma(z) = \frac{1}{1+e^{-z}}$ The sigmoid function varies most for input values around zero it is important that, when using gradient descent, the values of the different features are also in that range. If not, the gradient will be zero (or at least very small) and the optimization will take a very long time! Therefore it is common practice to scale the features.

Feature scaling

One approach is the following:

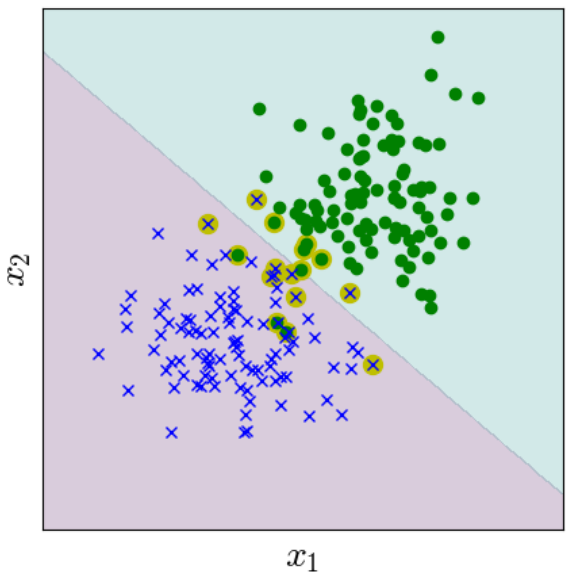
$$\mathbf{x}_{sc} = \frac{\mathbf{x} - \mu}{\sigma_{std}} \tag{35}$$

Where $\mu = \frac{1}{N+1} \sum_{i=0}^{N} x_i$ is the mean and $\sigma_{std} = \sqrt{\frac{1}{N+1} \|\mathbf{x} - \mu\|}$ the standard deviation. RESULT: all features have a mean of zero and a standard deviation of 1.

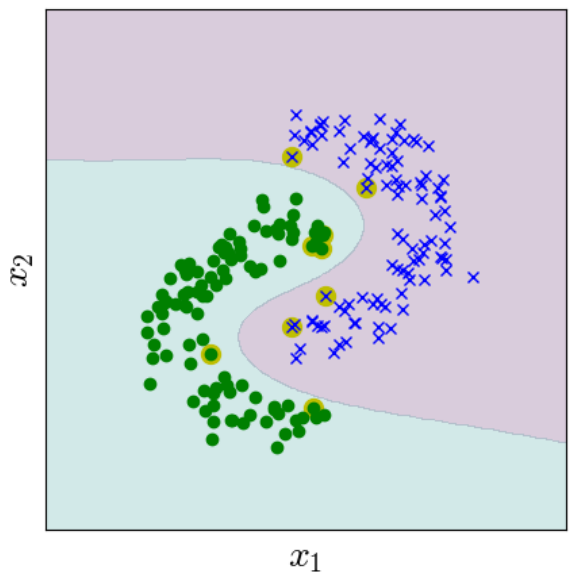
Not linearly separable problems

What if the data is not linearly separable?

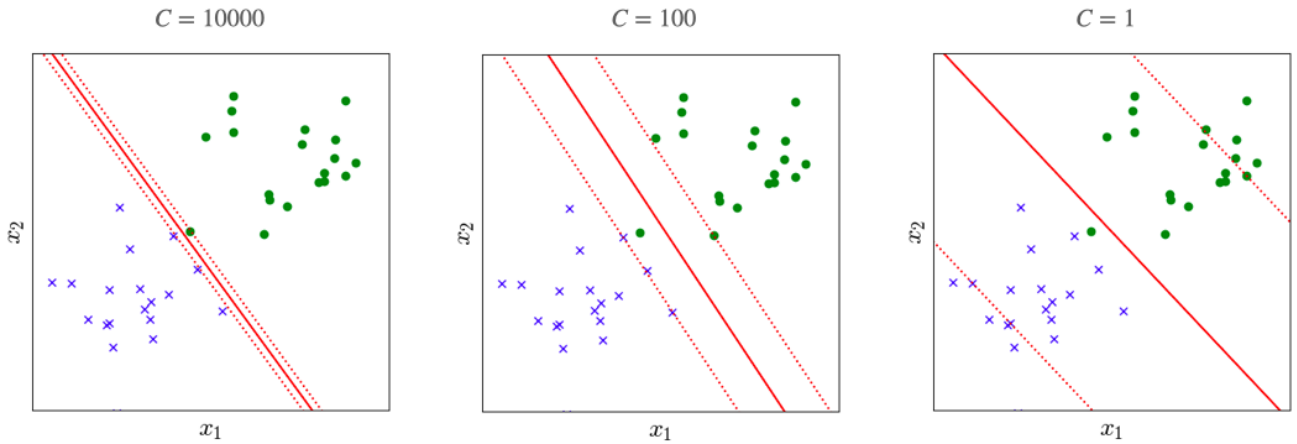
A. Allow misclassifications



B. Include more features (e.g. polynomial)



A. Allow misclassifications: Hinge loss function for soft margin



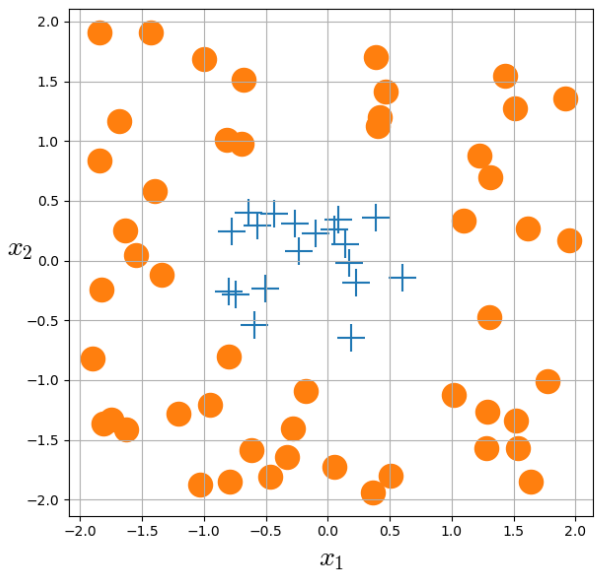
Minimize:

$$L(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{w}\|^2 + C\frac{1}{M}\sum_{m=0}^{M-1}\max\left(0, 1 - y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b)\right) \tag{36}$$

For $C \to \infty$ we have the hard margin loss function.

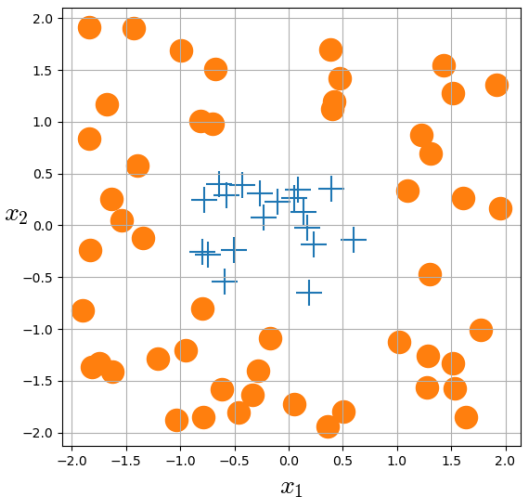
Polynomial features

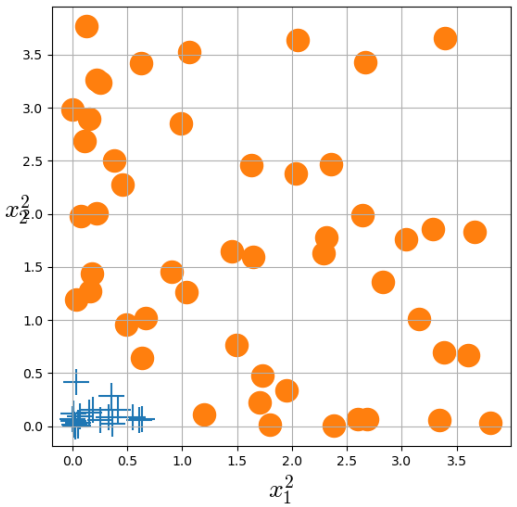
Data is not always linearly separable.



Polynomial features

For example use: $\mathbf{x} = [1, x_1^2, x_2^2]^T$





In general you can add many polynomial (or other) terms:

$$\mathbf{x} = [1, x_1, x_2, x_1^2, x_1 x_2, x_2^2, x_1^3, \cdots]^T$$

B. Include more polynomial features

Let's say we have a feature vector:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{37}$$

We can map this into a different feature *space* by e.g. computing all polynomial terms of degree 2 using a function $\phi(\mathbf{x})$:

$$\phi(\mathbf{x}) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \tag{38}$$

Classification: $\mathbf{x}_{new} \rightarrow \text{sgn}(\mathbf{w}^T\phi(\mathbf{x}_{new}) + b)$

However, this becomes computational expensive when the number of features N becomes very large.

B. Include more polynomial features

To solve this constrained optimization problem, we introduce Lagrange multipliers a_n:

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{2} \|\mathbf{w}\|^2 + \sum_{m=1}^{M} a_m \left(1 - y^{(m)}(\mathbf{w}^T \phi(\mathbf{x}^{(m)}) + b)\right) \tag{39}$$

Which leads to the **dual representation** (see Bishop Ch.7). minimize:

$$\tilde{L}(\mathbf{a}) = \sum_{m=1}^{M} a_m - \frac{1}{2} \sum_{m=1}^{M} \sum_{n=1}^{M} a_m a_n y^{(m)} y^{(n)} \mathrm{K}(\phi(\mathbf{x}^{(m)}), \phi(\mathbf{x}^{(n)})) \tag{40}$$

subject to the constraints:

$$a_m \geq 0, \quad \sum_{m=1}^{M} a_m y^{(m)} = 0, \quad m = 1, \ldots, M, \tag{41}$$

and with **kernel** function: $\mathrm{K}(\phi(\mathbf{x}^{(n)}), \phi(\mathbf{x}^{(m)})) = \mathbf{x}^{(n)^{\mathrm{T}}} \cdot \mathbf{x}^{(m)}$

B. Include more polynomial features

Which leads to the **dual representation** (see Bishop Ch.7).
minimize:

$$\tilde{L}(\mathbf{a}) = \sum_{m=1}^{M} a_m - \frac{1}{2} \sum_{m=1}^{M} \sum_{n=1}^{M} a_m a_n y^{(m)} y^{(n)} \mathrm{K}(\phi(\mathbf{x}^{(m)}), \phi(\mathbf{x}^{(n)})) \quad (42)$$

subject to the constraints:

$$a_m \geq 0, \quad \sum_{m=1}^{M} a_m y^{(m)} = 0, \quad m = 1, \ldots, M, \quad (43)$$

Classification: $\mathbf{x}_{new} \rightarrow \mathrm{sgn}\left(\sum_{m=1}^{M} a_m y^{(m)} \mathrm{K}\left(\mathbf{x}_{new}, \mathbf{x}^{(m)}\right) + b\right)$
Sparsification: $a_m = 0$ except for the support vectors.

B. Include more polynomial features

Advantage of the **dual representation** is that we went from an optimization problem involving N variables to one with M variables AND we can use **kernels**!

$$\begin{aligned}\phi(\mathbf{x}^{(1)}) \cdot \phi(\mathbf{x}^{(2)}) &= \begin{bmatrix} (x_1^{(1)})^2 \\ \sqrt{2}x_1^{(1)}x_2^{(1)} \\ (x_2^{(1)})^2 \end{bmatrix} \cdot \begin{bmatrix} (x_1^{(2)})^2 \\ \sqrt{2}x_1^{(2)}x_2^{(2)} \\ (x_2^{(2)})^2 \end{bmatrix} \\ &= (x_1^{(1)}x_1^{(2)})^2 + 2(x_1^{(1)}x_1^{(2)})(x_2^{(1)}x_2^{(2)}) + (x_2^{(1)}x_2^{(2)})^2 \\ &= (x_1^{(1)}x_1^{(2)} + x_2^{(1)}x_2^{(2)})^2 \\ &= (\mathbf{x}^{(1)} \cdot \mathbf{x}^{(2)})^2 \end{aligned} \tag{44}$$

B. Kernel: Include more polynomial features at low cost

$$
\begin{aligned}
(\mathbf{x}^{(1)} \cdot \mathbf{x}^{(2)})^4 &= \\
(x_1^{(1)}x_1^{(2)} + x_2^{(1)}x_2^{(2)})^4 &= \\
\left(x_1^{(1)}x_1^{(2)} + x_2^{(1)}x_2^{(2)}\right)^2 \left(x_1^{(1)}x_1^{(2)} + x_2^{(1)}x_2^{(2)}\right)^2 &= \\
\left((x_1^{(1)}x_1^{(2)})^2 + 2(x_1^{(1)}x_1^{(2)})(x_2^{(1)}x_2^{(2)}) + (x_2^{(1)}x_2^{(2)})^2\right) \left((x_1^{(1)}x_1^{(2)})^2 + 2(x_1^{(1)}x_1^{(2)})(x_2^{(1)}x_2^{(2)}) + (x_2^{(1)}x_2^{(2)})^2\right) &= \\
\begin{bmatrix} (x_1^{(1)})^4 \\ 2(x_1^{(1)}x_2^{(1)})^2 \\ (x_2^{(1)})^4 \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} (x_1^{(2)})^4 \\ 2(x_1^{(2)}x_2^{(2)})^2 \\ (x_2^{(2)})^4 \\ \vdots \end{bmatrix} &
\end{aligned}
$$

And what to think about $(\mathbf{x}^{(1)} \cdot \mathbf{x}^{(2)})^k$ for $k \to \infty$?

'kernel trick'

B. Kernel: Include more polynomial features at low cost

And what to think about $(\mathbf{x}^{(1)} \cdot \mathbf{x}^{(2)})^k$ for $k \to \infty$?
The widely used Gaussian kernel is an example of a distance ($d = \|\mathbf{d}_{ij}\| = \left\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right\|$) based kernel that can be formulated as a *polynomial kernel of infinite degree*:

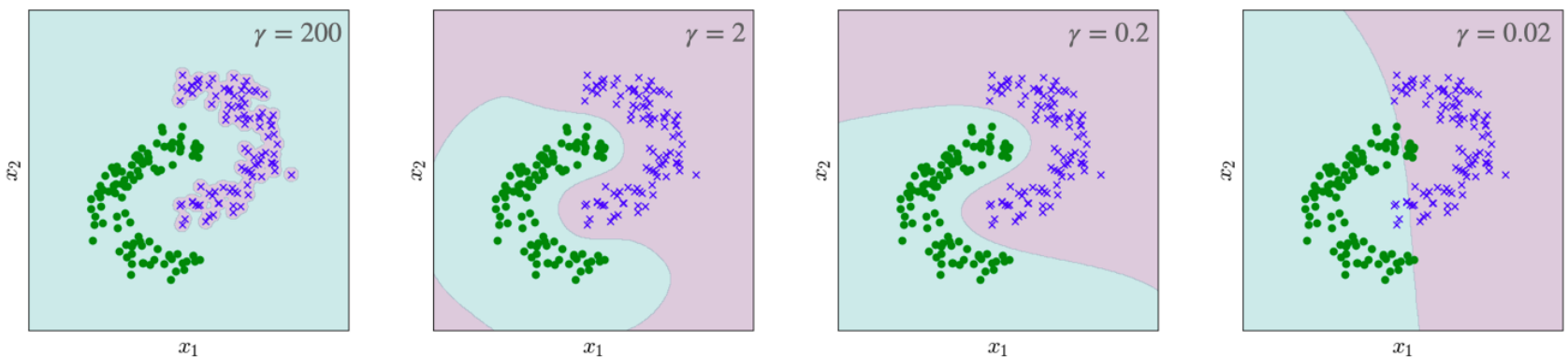
$$\mathrm{K}\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left(-\gamma \left\|\mathbf{d}_{ij}\right\|^2\right) =$$

$$\exp\left(-\gamma\, \mathbf{d}_{ij}^{\mathrm{T}} \cdot \mathbf{d}_{ij}\right) := \sum_{k=0}^{\infty} \frac{\left(-\gamma\, \mathbf{d}_{ij}^{\mathrm{T}} \cdot \mathbf{d}_{ij}\right)^k}{k!}. \tag{45}$$

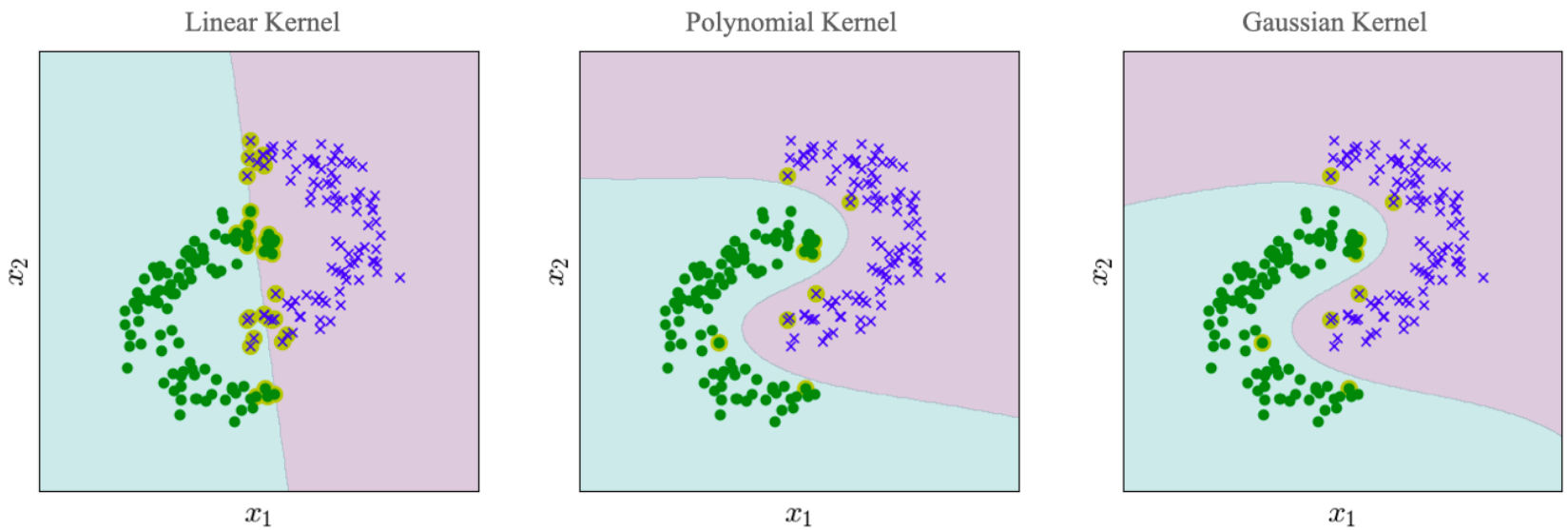
You can interpret the kernel as a **similarity measure**. Its maximum value (1) is obtained for the case $i = j$, that is the two feature vectors are identical.

B. Gaussian Kernel

$$\mathrm{K}\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left(-\gamma\, \mathbf{d}_{ij}^{\mathrm{T}} \cdot \mathbf{d}_{ij}\right)$$

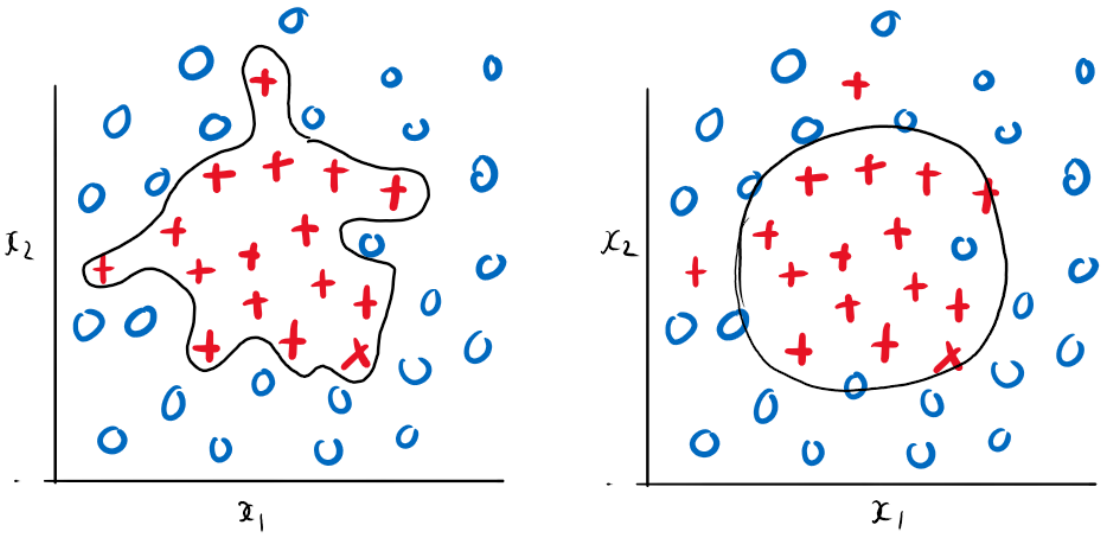


B. Other Kernels



Regularisation

The problem of over-fitting



To reduce overfitting, add an additional constrain that suppresses large weight values θ_i. Example Ridge Regression (or Tikhonov regularization):

$$L = \frac{1}{M}\frac{1}{2}\left\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right\|^2 + \frac{\lambda}{2}\left\|\boldsymbol{\theta}\right\|^2 \tag{46}$$

Introduction

What is a Neural Network (NN)?

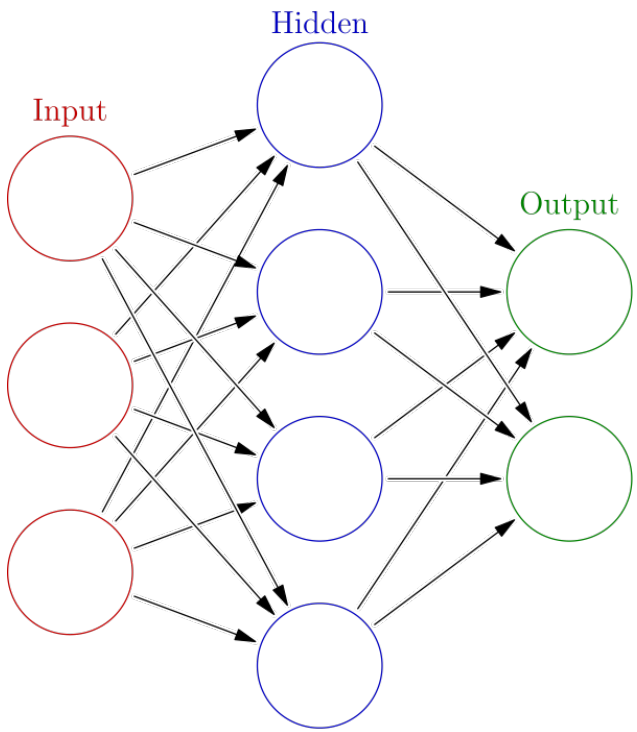
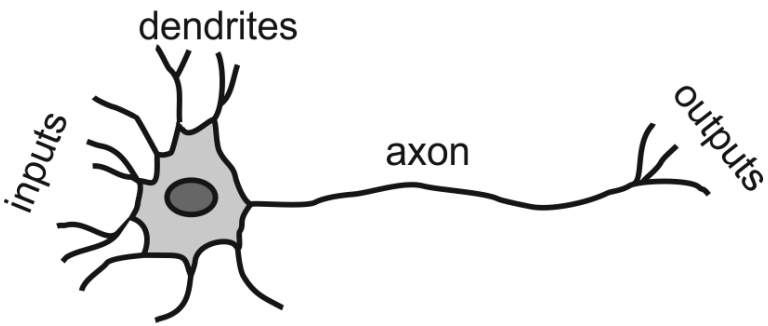
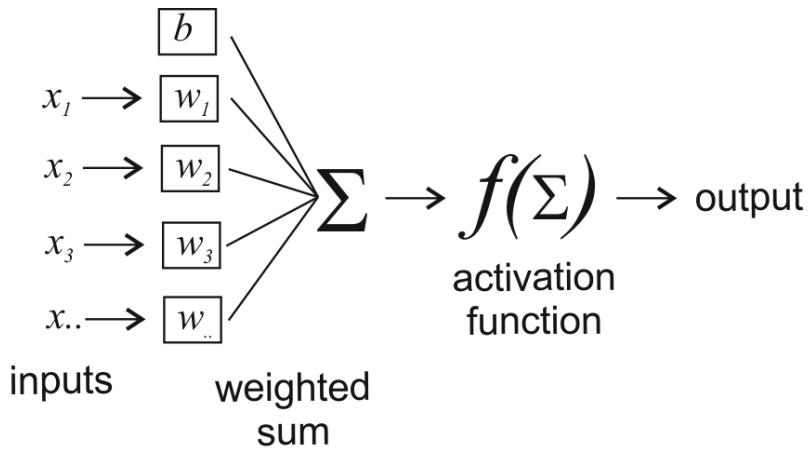


Figure: Glosser.ca, CC BY-SA 3.0 via Wikimedia Commons

Neurons

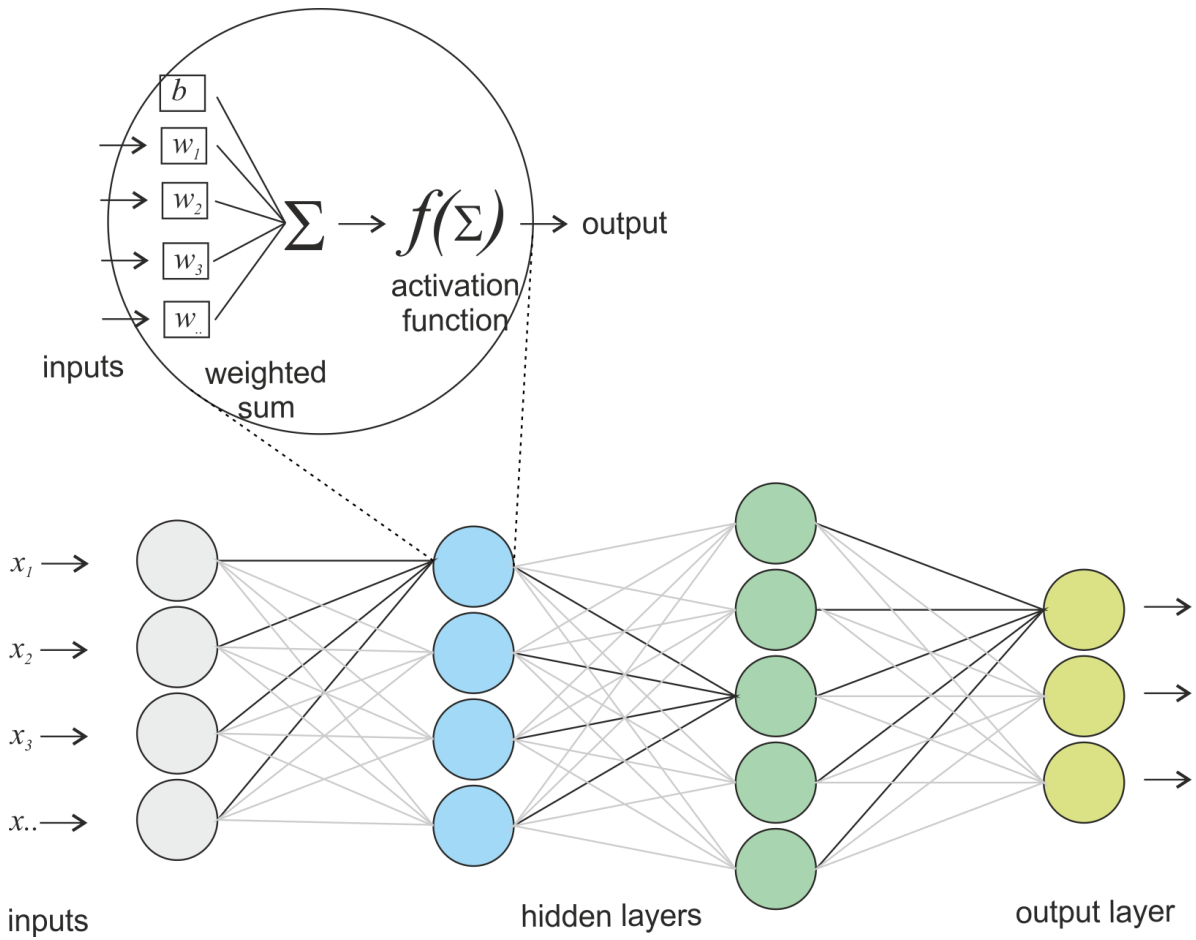


biological neuron



artifical neuron

Neural Network (NN)



Define the NN

The layer index is denoted with l. Our NN has an input layer ($l = 0$), an output layer ($l = L$) and hidden layers ($1 \leq l \leq L - 1$).
For each layer l we have:

- A number of nodes: n_l
- An output (column) vector: $\mathbf{a}^l \in \mathbb{R}^{n_l \times 1}$
- A weight matrix: $\mathbf{W}^l \in \mathbb{R}^{n_l \times n_{l-1}}$
- A bias (column) vector: $\mathbf{b}^l \in \mathbb{R}^{n_l \times 1}$
- An activation function: σ^l

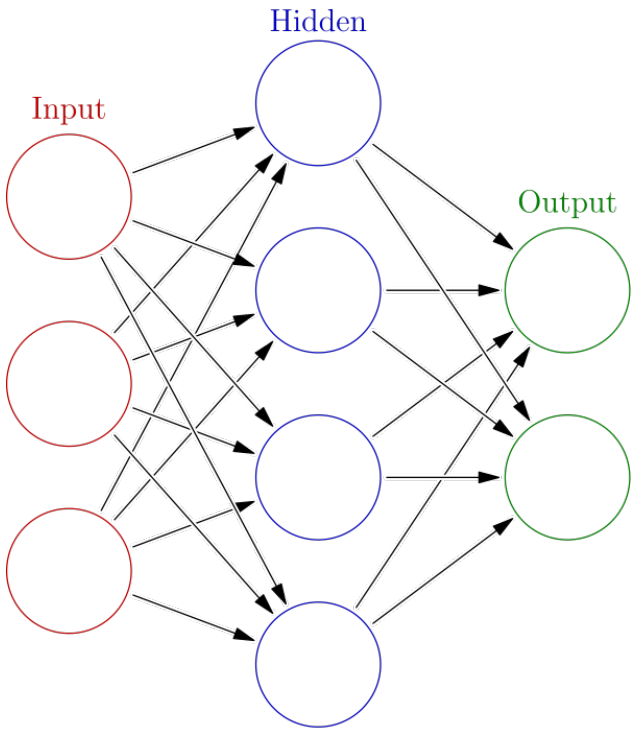


Figure: Glosser.ca, CC BY-SA 3.0 via Wikimedia Commons

Forward Pass, ie. use the (trained) NN to make a prediction

For the input layer:

$$\mathbf{a}^0 = \mathbf{x} \tag{47}$$

Forward Pass, ie. use the (trained) NN to make a prediction

For the input layer:

$$\mathbf{a}^0 = \mathbf{x} \tag{48}$$

For the hidden layer(s) ($1 \leq l \leq L$):

$$\begin{aligned} \mathbf{z}^l &= \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \\ \mathbf{a}^l &= \sigma^l\left(\mathbf{z}^l\right) \end{aligned} \tag{49}$$

Forward Pass, ie. use the (trained) NN to make a prediction

For the input layer:

$$\mathbf{a}^0 = \mathbf{x} \tag{50}$$

For the hidden layer(s) ($1 \leq l \leq L$):

$$\begin{aligned} \mathbf{z}^l &= \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \\ \mathbf{a}^l &= \sigma^l\left(\mathbf{z}^l\right) \end{aligned} \tag{51}$$

The above equations can also be written in component form as:

$$\begin{aligned} z_i^l &= \sum_{j=1}^{n_{l-1}} W_{ij}^l a_j^{l-1} + b_i^l \\ a_i^l &= \sigma^l\left(z_i^l\right) \end{aligned} \tag{52}$$

Note, no element-independent mapping for the Softmax, ie. $\mathbf{a}^l = \sigma^l(\mathbf{z})$.

Possible activation functions $\sigma(\mathbf{z})$

There are a few commonly used activation functions:

$$\text{Sigmoid} : \sigma_i(\mathbf{z}) = \frac{1}{1 + \exp(-\mathbf{z}_i)}$$

$$\text{ReLu} : \sigma_i(\mathbf{z}) = \max(0, \mathbf{z}_i)$$

$$\text{Tanh} : \sigma_i(\mathbf{z}) = \tanh(\mathbf{z}_i)$$

$$\text{Softmax} : \sigma_i(\mathbf{z}) = \frac{\exp(\mathbf{z}_i)}{\sum_{i=1}^{n_L} \exp(\mathbf{z}_i)}$$

In the Softmax, you might recognise similarity with the Boltzmann distribution: $p_i = e^{-\epsilon_i / k_B T} / \sum_i e^{-\epsilon_i / k_B T}$

Train the NN: Loss function

To train the NN we need a measure of how well the current NN performs. Two commonly applied loss functions that appear somewhat *naturally* are the:

- Quadratic (least-squares) loss function.

$$L_Q\left(\mathbf{W}^1, \mathbf{W}^2, ..\mathbf{W}^L, \mathbf{b}^1, \mathbf{b}^2, ..\mathbf{b}^L\right) = \frac{1}{M}\sum_{m=1}^{M}\left\|\mathbf{a}^L(\mathbf{x}^{(m)}) - \mathbf{y}^{(m)}\right\|^2$$

- The cross-entropy loss function (requires probabilities for label and prediction).

$$L_{CE}\left(\mathbf{W}^1, \mathbf{W}^2, ..\mathbf{W}^L, \mathbf{b}^1, \mathbf{b}^2, ..\mathbf{b}^L\right) = -\frac{1}{M}\sum_{m=1}^{M}\mathbf{y}^{(m)} \cdot \ln \mathbf{a}^L(\mathbf{x}^{(m)})$$

One-hot (probability) representation

Example: Recognition of handwritten numbers 1,2,3,4:
label number 1 = [1 0 0 0], state =0
label number 2 = [0 1 0 0], state =1
label number 3 = [0 0 1 0], state =2
label number 4 = [0 0 0 1], state =3

One-hot (probability) representation

Example: Recognition of handwritten numbers 1,2,3,4:
label number 4 = [0 0 0 1]

test output network $\mathbf{z}$ =[a b c d]
$\rightarrow$ Softmax : $\sigma_i(\mathbf{z}) = \frac{\exp(\mathbf{z}_i)}{\sum_{i=1}^{n_L} \exp(\mathbf{z}_i)}$ =[0.15 0.01 0.04 0.80]

prediction = arg.max([0.15 0.01 0.04 0.80]) = state 3 $\rightarrow$ number 4

One-hot (probability) representation

Example: Recognition of handwritten numbers 1,2,3,4:
label number 4 = [0 0 0 1]

prediction = arg.max([0.15 0.01 0.04 0.80]) = state 3 $\rightarrow$ number 4

$$L_{CE} = -\frac{1}{M}\sum_{m=1}^{M} \mathbf{y}^{(m)} \cdot \ln \mathbf{a}^{L}(\mathbf{x}^{(m)})$$

$L_{CE} = -[0\ 0\ 0\ 1] \cdot \ln([0.15\ 0.01\ 0.04\ 0.80])$
$L_{CE} = -[0\ 0\ 0\ 1] \cdot [-1.9\ -4.6\ -3.2\ -0.22])$
$L_{CE} = 0 * 1.9 + 0 * 4.6 + 0 * 3.2 + 1 * 0.22 = 0.22$

Train the NN: Gradient Decent

We want to minimize the loss function. For both L_Q and L_{CE} the perfect fit will result in $L = 0$.
In the stochastic gradient approach all parameters (weights and biases) need to be updated:

$$W_{ij}^l \to W_{ij}^l - \eta \sum_{batch} \frac{\partial L}{\partial W_{ij}^l}$$

$$b_i^l \to b_i^l - \eta \sum_{batch} \frac{\partial L}{\partial b_i^l}$$

Where η is the *learning rate*. The gradients can be computed by **back propagation**.

Train the NN: Back propagation

A great advantage, combining linearity of the NN with the *chain rule*

In case of a 1-layer L_Q NN: $\mathbf{a} = \sigma(\mathbf{z}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$
$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = (\mathbf{a} - \mathbf{y})(\sigma'(\mathbf{z}))(\mathbf{x})$.

In a general L layer NN:

$$\frac{\partial L}{\partial \mathbf{z}^L} = \begin{cases} \frac{\mathrm{d}\sigma^L}{\mathrm{d}\mathbf{z}^L} \circ \left(\mathbf{a}^L - \mathbf{y}\right) & \left(L_Q \text{ and all element-independent activation functions} \right) \\ \mathbf{a}^L - \mathbf{y} & \left(L_{CE} \text{ and Softmax} \right) \end{cases} \tag{1}$$

Train the NN: Back propagation

A great advantage, combining linearity of the NN with the *chain rule*.

$$\frac{\partial L}{\partial \mathbf{z}^L} = \begin{cases} \frac{\mathrm{d}\sigma^L}{\mathrm{d}\mathbf{z}^L} \circ \left(\mathbf{a}^L - \mathbf{y}\right) & \left(L_Q \text{ and all element-independent activation functions}\right) \\ \mathbf{a}^L - \mathbf{y} & \left(L_{CE} \text{ and Softmax}\right) \end{cases} \tag{1}$$

$$\frac{\partial L}{\partial \mathbf{z}^l} = \frac{\mathrm{d}\sigma^l}{\mathrm{d}\mathbf{z}^l} \circ (\mathbf{W}^{l+1})^T \frac{\partial L}{\partial \mathbf{z}^{l+1}} \qquad \text{for } 1 \leq l \leq L-1 \tag{2}$$

Train the NN: Back propagation

A great advantage, combining linearity of the NN with the *chain rule*.

$$\frac{\partial L}{\partial \mathbf{z}^L} = \begin{cases} \frac{\mathrm{d}\sigma^L}{\mathrm{d}\mathbf{z}^L} \circ \left(\mathbf{a}^L - \mathbf{y}\right) & \left(L_Q \text{ and all element-independent activation functions}\right) \\ \mathbf{a}^L - \mathbf{y} & \left(L_{CE} \text{ and Softmax}\right) \end{cases} \tag{1}$$

$$\frac{\partial L}{\partial \mathbf{z}^l} = \frac{\mathrm{d}\sigma^l}{\mathrm{d}\mathbf{z}^l} \circ (\mathbf{W}^{l+1})^T \frac{\partial L}{\partial \mathbf{z}^{l+1}} \qquad \text{for } 1 \leq l \leq L-1 \tag{2}$$

$$\frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \mathbf{z}^l} (\mathbf{a}^{l-1})^T \qquad \text{for } 1 \leq l \leq L \tag{3}$$

Train the NN: Back propagation

A great advantage, combining linearity of the NN with the *chain rule*.

$$\frac{\partial L}{\partial \mathbf{z}^L} = \begin{cases} \frac{\mathrm{d}\sigma^L}{\mathrm{d}\mathbf{z}^L} \circ \left(\mathbf{a}^L - \mathbf{y}\right) & \left(L_Q \text{ and all element-independent activation functions}\right) \\ \mathbf{a}^L - \mathbf{y} & \left(L_{CE} \text{ and Softmax}\right) \end{cases} \tag{1}$$

$$\frac{\partial L}{\partial \mathbf{z}^l} = \frac{\mathrm{d}\sigma^l}{\mathrm{d}\mathbf{z}^l} \circ (\mathbf{W}^{l+1})^T \frac{\partial L}{\partial \mathbf{z}^{l+1}} \quad \text{for } 1 \leq l \leq L-1 \tag{2}$$

$$\frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \mathbf{z}^l} (\mathbf{a}^{l-1})^T \quad \text{for } 1 \leq l \leq L \tag{3}$$

$$\frac{\partial L}{\partial \mathbf{b}^l} = \frac{\partial L}{\partial \mathbf{z}^l} \quad \text{for } 1 \leq l \leq L \tag{4}$$

Train the NN: Back propagation

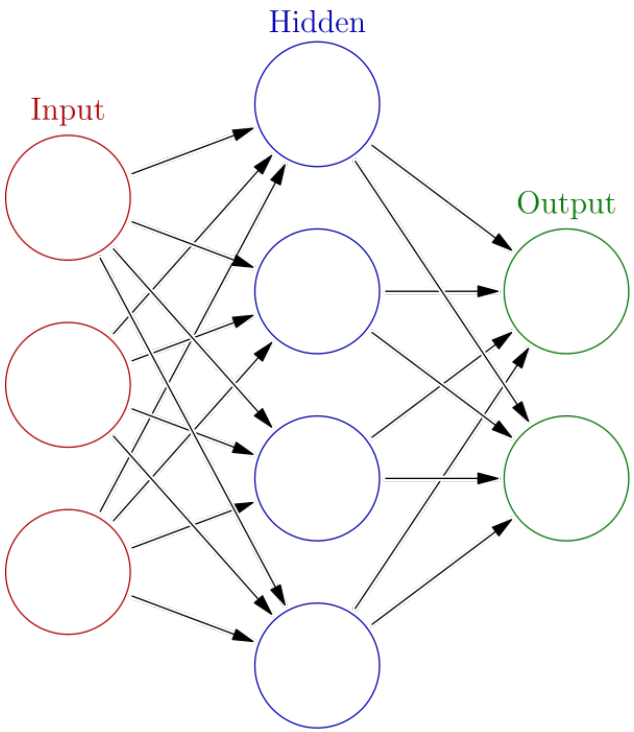


Figure: Glosser.ca, CC BY-SA 3.0 via Wikimedia Commons

$$\frac{\partial L}{\partial \mathbf{z}^L} = \begin{cases} \frac{\mathrm{d}\sigma^L}{\mathrm{d}\mathbf{z}^L} \circ \left(\mathbf{a}^L - \mathbf{y}\right) & \left(L_Q \text{ and all element-indepen} \right. \\ \mathbf{a}^L - \mathbf{y} & \left(L_{CE} \text{ and } \right. \end{cases} \tag{1}$$

$$\frac{\partial L}{\partial \mathbf{z}^l} = \frac{\mathrm{d}\sigma^l}{\mathrm{d}\mathbf{z}^l} \circ (\mathbf{W}^{l+1})^T \frac{\partial L}{\partial \mathbf{z}^{l+1}} \qquad \text{for } 1 \leq l \leq \tag{2}$$

$$\frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \mathbf{z}^l} (\mathbf{a}^{l-1})^T \qquad \text{for } 1 \leq l \leq \tag{3}$$

$$\frac{\partial L}{\partial \mathbf{b}^l} = \frac{\partial L}{\partial \mathbf{z}^l} \qquad \text{for } 1 \leq l \leq \tag{4}$$

NN training strategy

Split your valuable data up in three sets:

1. **A training set**. (Large set)
2. **A validation set**. (Small set)
3. **A test set**. (Small set)

Again, there is no golden rule, but try 60/20/20 % to start off.

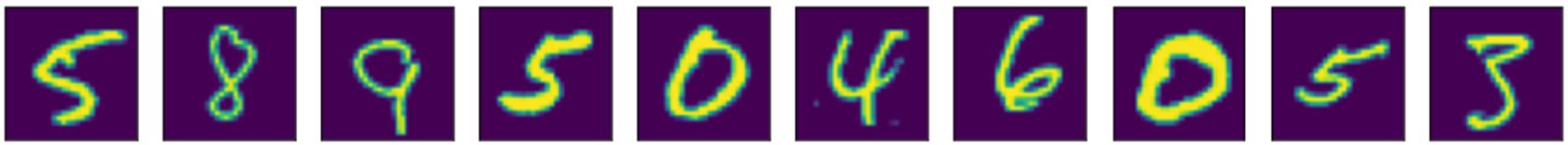
NN training strategy

Now we can start training:

1. **create batches**. Divide the data in a batches of equal size.
2. **epoch**. For each batch do:
   1. **forward pass**. Calculate the output of the NN ($\mathbf{a}^L$) for each data point in the batch.
   2. **backward pass**. Compute the derivatives $\partial L/\partial \mathbf{W}^l$ and $\partial L/\partial \mathbf{b}^l$ for all layers and sum over all data points in the batch.
   3. **update weights**. Compute the new weights and biases.
3. **compute loss**. Compute the loss of the training and validation set.
4. **Repeat**. Repeat steps 2 through 4 until convergence.
5. **test**. Evaluate the performance of the trained NN by computing the loss of the the test set.

Python Notebook (option 1)

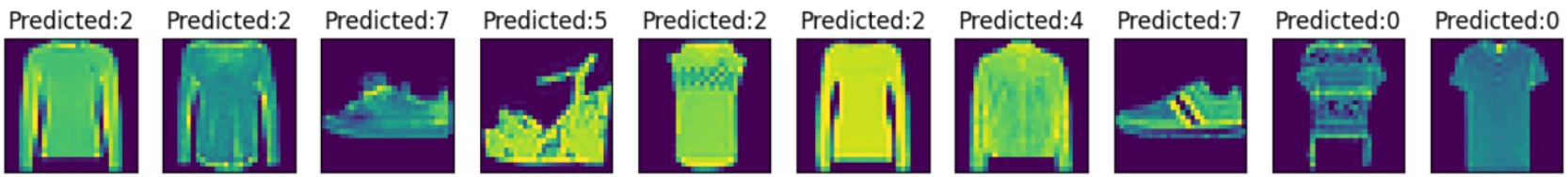
Notebook: `neural_network_part2_student.ipynb`
The MNIST dataset: 70000 images (28×28) of handwritten digits.



Build a NN that correctly classifies the images ($> 95\%$).

Python Notebook(option 2)

Notebook: `neural_network_part2_student.ipynb`
The Fashion-MNIST dataset: 70000 images (28×28) of handwritten digits.



Build a NN that correctly classifies the images ($> 95\%$).

Points of attention

- Carefully check definition of in- and output shapes and of weights matrices and bias vectors etc.
- Train the network using gradient descent using *batches* (see also notebook)
- Normalization/scaling of the data is very important