Introduction

Patrick van der Smagt

March 2023

You have data from a stochastic system:

- from an airport camera,
- from a camera seeing an insect flying, a body suit on a human, . . .
- weather observations, stock market data,
- a tactile sensor on a robot hand, etc.

and you want to use these data to

- detect suspicious behaviour,
- classify movement,
- predict future data,
- control a system

How?

The description is simple, in the form a model of your stochastic process:

$$y = f(x)$$

where y, x are vectors, and $f()$ is the underlying model.

Consider the above also in the special form

$$x_{t+1} = f(x_t)$$

where we use the Markov assumption.

Key question: how do we get $f()$?

approximating $f()$ through system knowledge

If, e,g., we find all red objects in HSV space
– or we compute the Cartesian end position of a robot arm by inverse kinematics
– or we compute radiation of an exploding supernova
– or we want to determine a plant by analysing stem, leaves, flowers
–
in each of these cases, we formalise expert knowledge in terms of a set of rules or decisions.

This is known classical control, expert systems, etc.

It is your best solution if your approximation ('model') $\hat{f}_\theta()$ to $f()$ can be described at your desired accuracy. The description of $\hat{f}()$ is obtained using innate knowledge of f. How do you know if your accuracy is good enough?

approximating $f()$ when system knowledge fails

If, e.g., you want to recognise faces, or a plant from a picture
– or want to create a programme to play go
– or want to find good grip positions on a random object
– or want to recognise a composer by listening to their music
–
in each of these cases, the number of cases is too large to exhaustively describe accurately enough.

We have no detailed knowledge of $f()$, and instead we use a general form for $\hat{f}()$ that can represent any function.

But both approaches are in principle the same:

1. create a parameterised model $\hat{f}(x, \theta)$
2. find best values for the parameters θ

How do we do the second step? Two options:

a. we make $\hat{f}$ behave as close as possible to f (e.g., maximise number of go games won)

b. we make $\hat{f}$ be as close as possible to f through $\min_\theta |\hat{f}_\theta(x) - f(x)|$. Failing f, we sample f in $\{x_i, y_i\}$ to $\min_\theta \sum_i |\hat{f}_\theta(x_i) - y_i|$ as a proxy. We can write this as $\max_\theta \prod_i p_\theta(y_i \mid x_i)$.

The basic problem that we study in probability theory:

Given a data generating process,
what are the properties of the outcomes?

The basic problem of statistics (or better statistical inference)
is the inverse of probability theory:

Given the outcomes,
what can we say about the process that generated the data?

Statistics uses the formal language of probability theory.

discriminative vs. generative models

The above describes *discriminative* models, which map inputs to outputs and are described by $p(y|x)$.

Often we are interested in *generative* models, represented by
$p(x, y)$ (if we have labels) or
$p(x)$ (if we have no labels).

We can still measure quality, through $\max_\theta \prod_i p_\theta(x_i)$.

Generative models can be used to generate new data, or to test the likelihood of a datum.

examples of discriminative models

This is nothing new. Typically, such models use

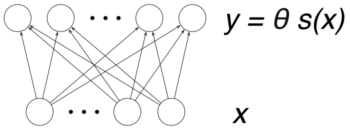
$$\hat{f}_\theta(x) = \sum_k \theta_k s_k(x)$$

which describes, for instance, Fourier transform, polynomials, etc.

This general form is known as *linear regression*, because the regression function is linear in the parameters θ.

This is important as we have a closed-form solution to find θ.

In the connectionism paradigm, the models can be drawn like this:



aka the perceptron

curse of dimensionality

Even if x is only 100-dimensional

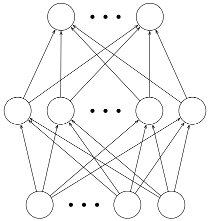
and you have a trillion data,

those data cover only 10^{-18} of the input space.

This can be very problematic for such models.

We need models that generalise further.

extending the connectionist idea



$y = s(\theta_2\, h)$

$h = s(\theta_1\, x)$

x

did you see the small trick I did?

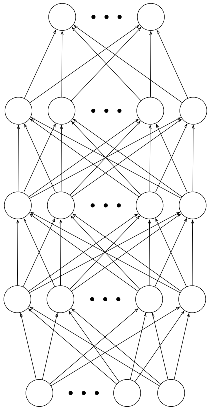
That's a great idea from the 1970s which suggests,

$$\hat{f}_\theta(x) = \sum_k \theta_k s \left(\sum_i \theta_l x \right)$$

and the neural network (= multi-layer perceptron) is born.

Problem: finding θ can no longer be done closed-form.
Gradient-based optimisation is the standard approach.

a deep neural network just has more layers



$y = s(\theta_4 h_3)$

$h_3 = s(\theta_3 h_2)$

$h_2 = s(\theta_2 h_1)$

$h_1 = s(\theta_1 x)$

x

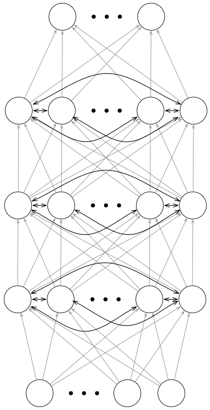
history of neural networks 1

1960s:	the linear perceptron learns from data (Rosenblatt et al)
1969:	the perceptron can't do XOR (Minsky & Papert)
1970–1980:	nonlinear networks trained with back-propagation (Linnainmaa; Dreyfus; Werbos; Rumelhart)
1990s:	one hidden layer can represent any (Borel-measurable) function
mid 90s:	NN's can't do everything / do not generalise / . . .
mid 90s:	support vector machines (SVMs) are great!
1995–2000:	SVM too slow / too many SVs

history of neural networks 2

2000–:	probabilistic models for machine learning (ML)
2006:	**deep neural networks**, trained with *Restricted Boltzmann Machines* and backprop
2009:	deep NNs can be trained with just BP, **much compute power** (GPU) and **enough data**
2011–:	recurrent neural networks resurrect for time-series modelling
2012–:	convolutional neural networks (CNN) start winning most vision benchmarks; recurrent neural networks applied to speech recognition in Android
2013–:	**probabilistic** NN (variance propagation; variational autoencoder)
2015–:	show cases in robotics, sensory processing, . . .
2017:	"Attention is all you need" . . .

a recurrent neural network has an internal state



$y = s(\theta_4\, h_3)$

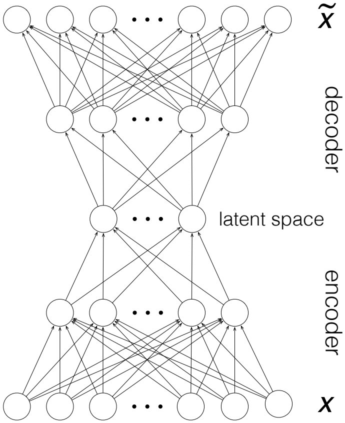
$h_3 = s(\theta_{31}\, h_2 + \theta_{32}\, h_3)$

$h_2 = s(\theta_{21}\, h_1 + \theta_{22}\, h_2)$

$h_1 = s(\theta_{11}\, x + \theta_{12}\, h_1)$

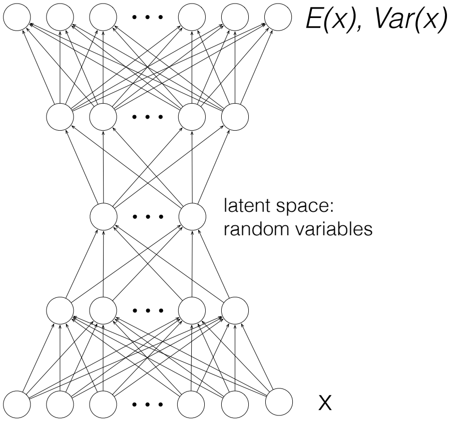
x

autoencoder: NN in a special form



$\text{Dim(latent space)} \ll \text{Dim}(x)$

VAE: probabilistic AE



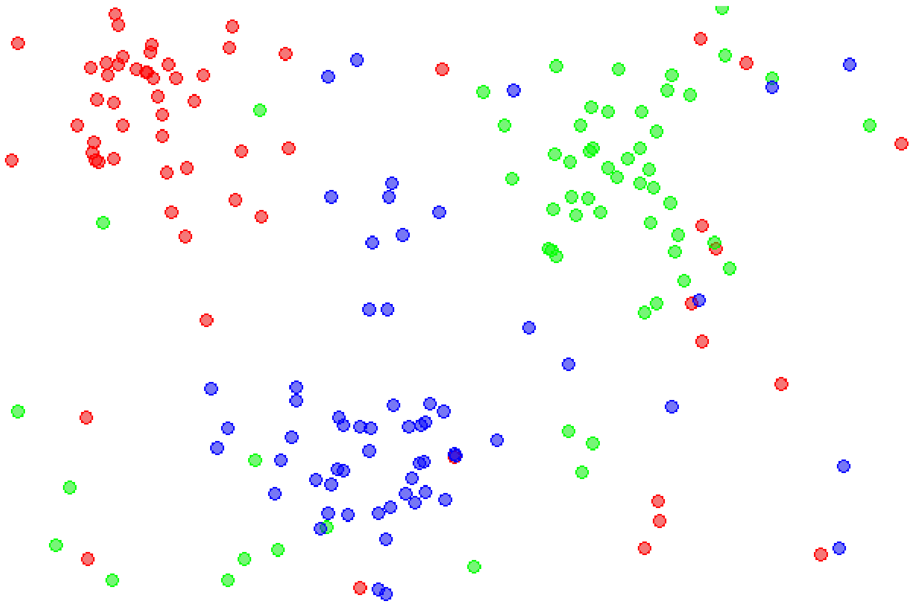
'nonlinear PCA'

start experimenting

Start the first notebook of this class on your machine

or use `https://colab.research.google.com/github/smagt/intro-to-ml/`

unsupervised learning: k-nearest neighbour

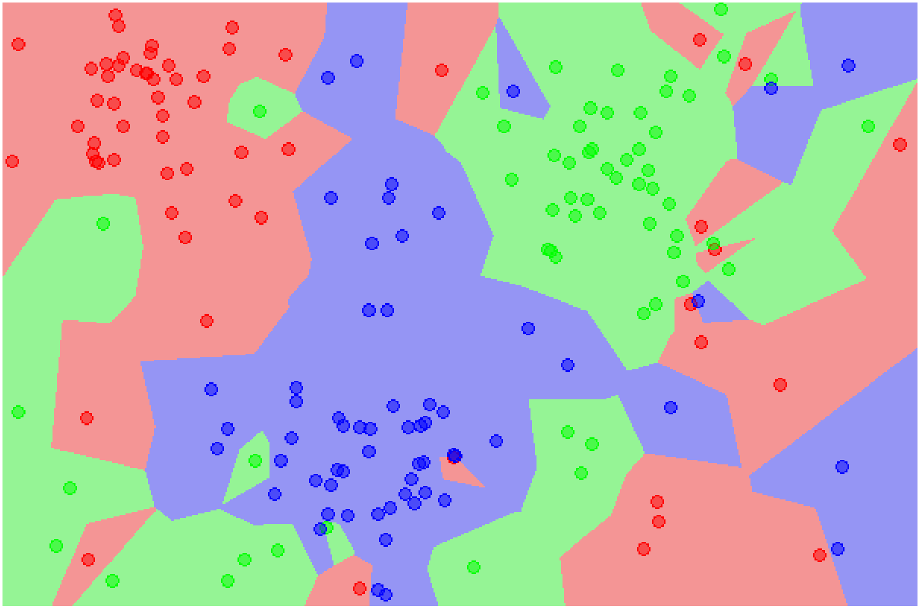


1-NN algorithm

Example: let's take $k = 1$

1. define a distance measure
2. determine the number of classes
3. for each new data point x:
    1. determine the distance to all other points
    2. find the nearest neighbours
4. the class c of the new data point is determined

1-NN algorithm: overfitting



k-NN equation

how can we describe this equation?

k-NN equation

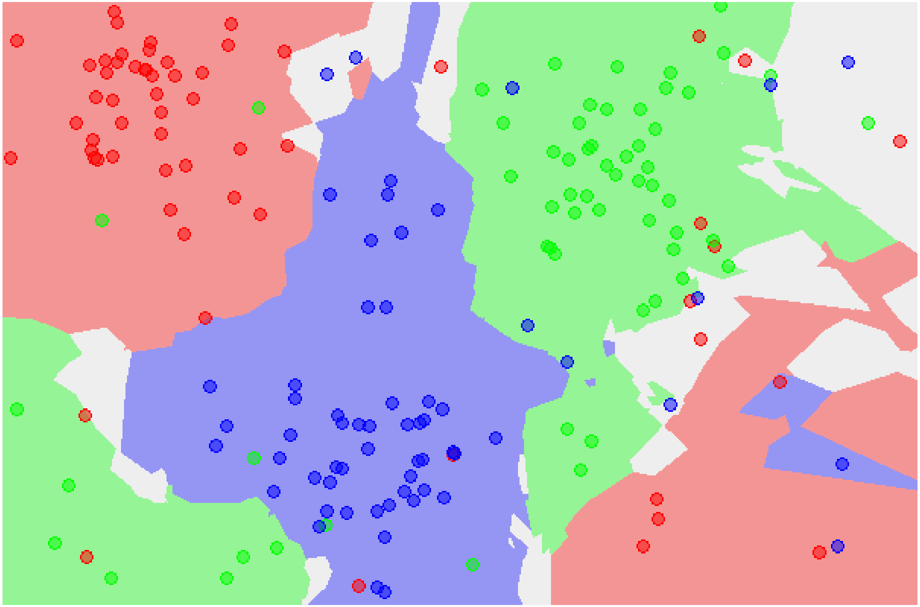
how can we describe this equation?

$$p(y = c \mid x) = \frac{1}{k} \sum_{i \in \text{neighbours}} \delta_{y_i c}$$

mit

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

4-NN algorithm



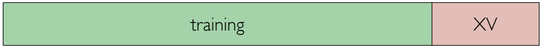
finding hyperparameters

k is a hyperparameter, the value of which determines the outcome. How can we optimise it?

finding hyperparameters

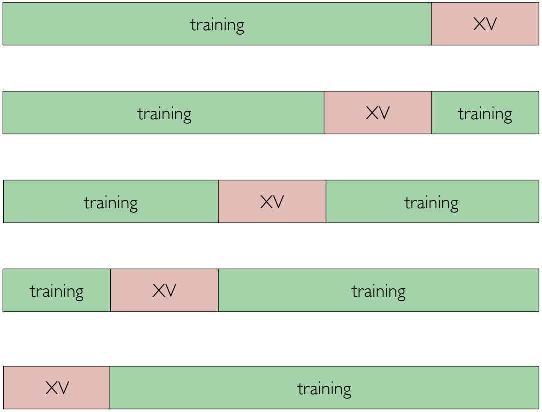
k is a hyperparameter, the value of which determines the outcome. How can we optimise it?

by cross validation



finding hyperparameters

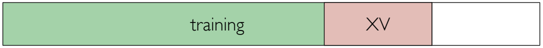
5-fold crossvalidation



finding hyperparameters

but a more honest method is:

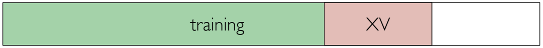
1. first learn



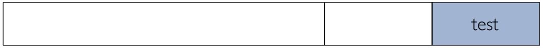
finding hyperparameters

but a more honest method is:

1. first learn



2. then test on new data



problems with kNN

- finding the right distance measure is difficult and strongly influences the result
- in high-dimensional spaces, distances stop making sense

  `https://stats.stackexchange.com/questions/99171/why-is-euclidean-distance-not-a-good-metric-in-high-dimensions`
- finding the neighbours is very expensive in high-dimensional spaces
- finding the neighbours is very expensive if many data exist