# Part 1: Introduction

**A few useful things to know about machine learning**

Joaquin Vanschoren, Eindhoven University of Technology

# Artificial Intelligence

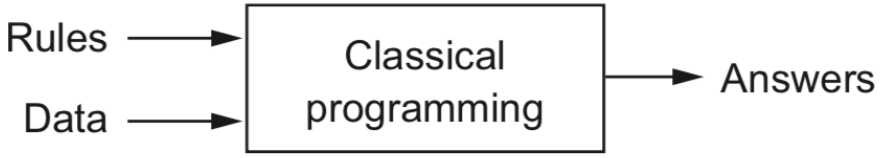
1950s: Can computers be made to 'think'?

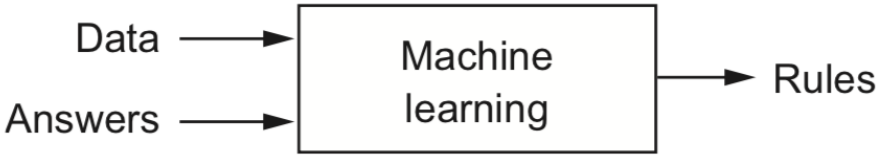
- automate intellectual tasks normally performed by humans
- encompasses learning, but also many other tasks (e.g. logic, planning,...)
- *symbolic AI*: programmed rules/algorithms for manipulating knowledge
    - Great for well-defined problems: chess, expert systems,...
    - Pervasively used today (e.g. chip design)
    - Hard for complex, fuzzy problems (e.g. images, text)

# Machine Learning

Are computers capable of learning and originality? Alan Turing: Yes!

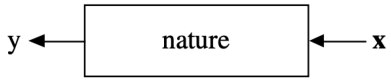
- Learn to perform a task T given experience (examples) E, always improving according to some metric M
- New programming paradigm
    - System is *trained* rather than explictly programmed
    - *Generalizes* from examples to find rules (models) to act/predict
- As more data becomes available, more ambitious problems can be tackled



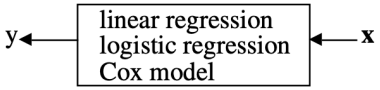



# Machine learning vs Statistics

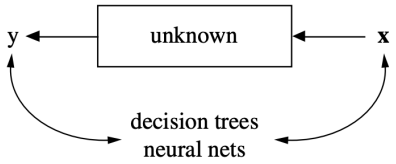
- Both aim to make predictions of natural phenomena:




- Statistics:
  - Help humans understand the world
  - Parametric: assume data is generated according to parametric model




- Machine learning:
  - Automate a task entirely (partially *replace* the human)
  - Assume that data generation process is unknown
  - Engineering-oriented, less (too little?) mathematical theory




See Breiman (2001): Statistical modelling: The two cultures

## Machine Learning success stories

- Search engines (e.g. Google)
- Recommender systems (e.g. Netflix)
- Automatic translation (e.g. Google Translate)
- Speech understanding (e.g. Siri, Alexa)
- Game playing (e.g. AlphaGo)
- Self-driving cars
- Personalized medicine
- Progress in all sciences: Genetics, astronomy, chemistry, neurology, physics,..

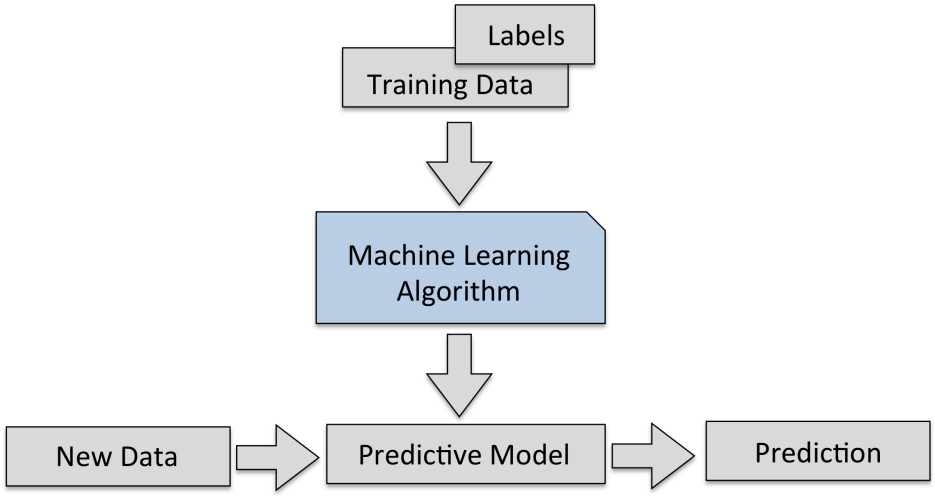
# Types of machine learning

- **Supervised Learning**: learn a *model* from labeled *training data* (ground truth)
    - Given a new input $X$, predict the right output $y$
    - Given images of cats and dogs, predict whether a new image is a cat or a dog
- **Unsupervised Learning**: explore the structure of the data to extract meaningful information
    - Given inputs $X$, find which ones are special, similar, anomalous, ...
- **Semi-Supervised Learning**: learn a model from (few) labeled and (many) unlabeled examples
    - Unlabeled examples add information about which new examples are likely to occur
- **Reinforcement Learning**: develop an agent that improves its performance based on interactions with the environment

Note: Practical ML systems can combine many types in one system.

# Supervised Machine Learning

- Learn a model from labeled training data, then make predictions
- Supervised: we know the correct/desired outcome (label)
- Subtypes: *classification* (predict a class) and *regression* (predict a numeric value)
- Most supervised algorithms that we will see can do both



## Classification

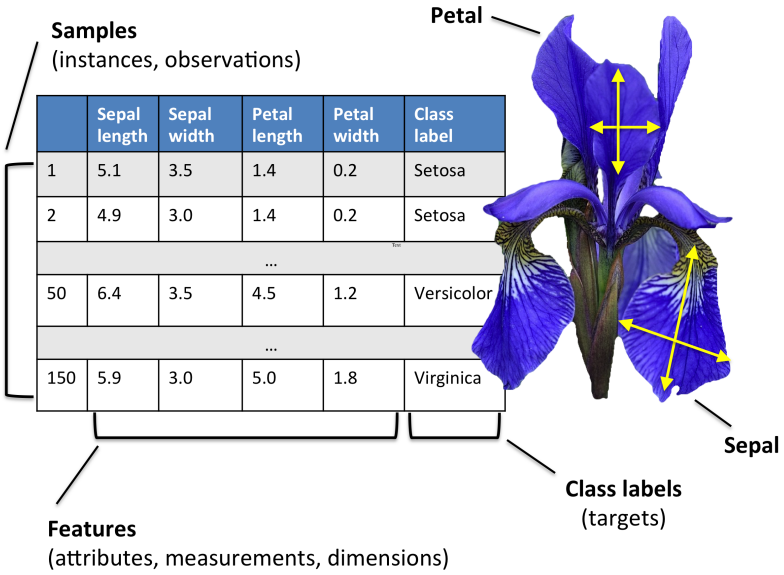
- Predict a *class label* (category), discrete and unordered
  - Can be *binary* (e.g. spam/not spam) or *multi-class* (e.g. letter recognition)
  - Many classifiers can return a *confidence* per class
- The predictions of the model yield a *decision boundary* separating the classes

**Example: Flower classification**

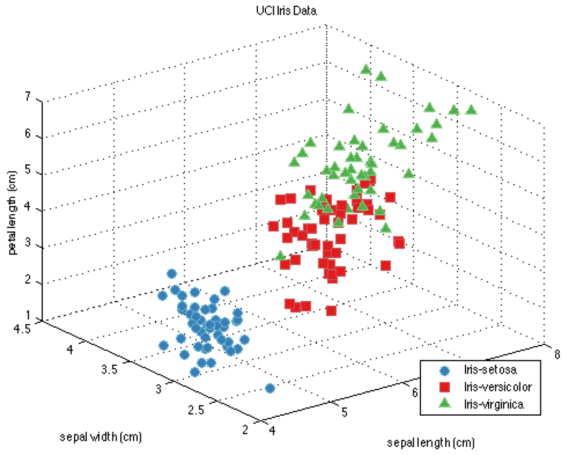
Classify types of Iris flowers (setosa, versicolor, or virginica). How would you do it?

## Representation: input features and labels

- We could take pictures and use them (pixel values) as inputs (-> Deep Learning)
- We can manually define a number of input features (variables), e.g. length and width of leaves
- Every `example' is a point in a (possibly high-dimensional) space




| | Sepal length | Sepal width | Petal length | Petal width | Class label |
|---|---|---|---|---|---|
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | Setosa |
| 2 | 4.9 | 3.0 | 1.4 | 0.2 | Setosa |
| ... | | | | | |
| 50 | 6.4 | 3.5 | 4.5 | 1.2 | Versicolor |
| ... | | | | | |
| 150 | 5.9 | 3.0 | 5.0 | 1.8 | Virginica |



## Regression

- Predict a continuous value, e.g. temperature
    - Target variable is numeric
    - Some algorithms can return a *confidence interval*
- Find the relationship between predictors and the target.
    - E.g. relationship between hours studied and final grade

# Unsupervised Machine Learning

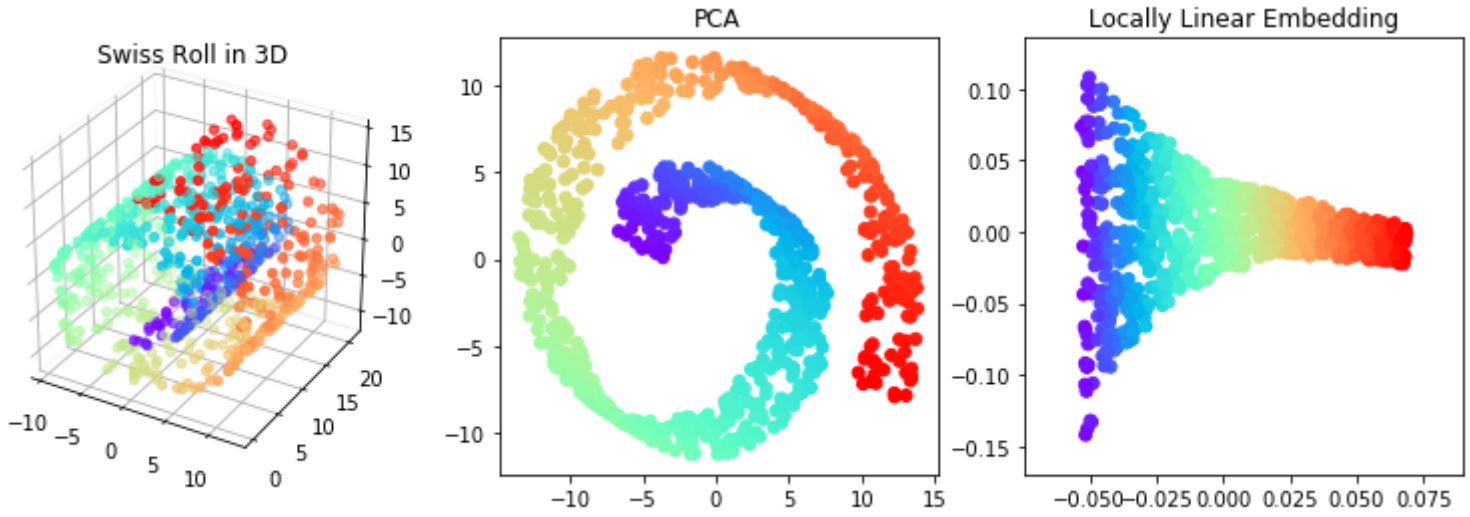
- Unlabeled data, or data with unknown structure
- Explore the structure of the data to extract information
- Many types, we'll just discuss two.

# Clustering

- Organize information into meaningful subgroups (clusters)
- Objects in cluster share certain degree of similarity (and dissimilarity to other clusters)
- Example: distinguish different types of customers

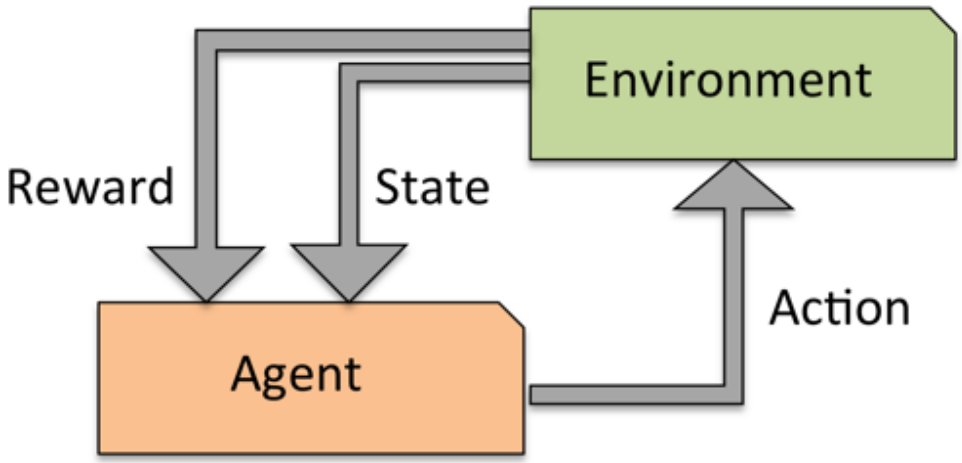
# Dimensionality reduction

- Data can be very high-dimensional and difficult to understand, learn from, store,...
- Dimensionality reduction can compress the data into fewer dimensions, while retaining most of the information
- Contrary to feature selection, the new features lose their (original) meaning
- The new representation can be a lot easier to model (and visualize)



# Reinforcement learning

- Develop an agent that improves its performance based on interactions with the environment
    - Example: games like Chess, Go,...
- Search a (large) space of actions and states
- *Reward function* defines how well a (series of) actions works
- Learn a series of actions (policy) that maximizes reward through exploration



# Learning = Representation + evaluation + optimization

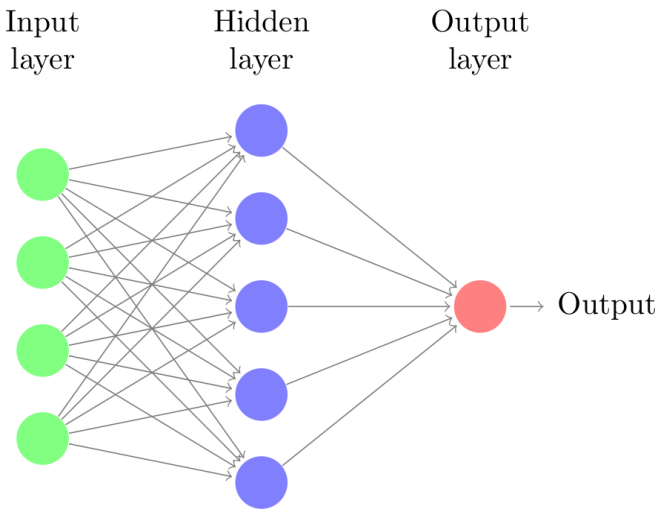
All machine learning algorithms consist of 3 components:

- **Representation**: A model must be represented in a formal language that the computer can handle
    - Defines the 'concepts' it can learn, the *hypothesis space*
    - E.g. a decision tree, neural network, set of annotated data points
- **Evaluation**: An *internal* way to choose one hypothesis over the other
    - Objective function, scoring function, loss function
    - E.g. Difference between correct output and predictions
- **Optimization**: An *efficient* way to search the hypothesis space
    - Start from simple hypothesis, extend (relax) if it doesn't fit the data
    - Defines speed of learning, number of optima,...
    - E.g. Gradient descent

A powerful/flexible model is only useful if it can also be optimized efficiently

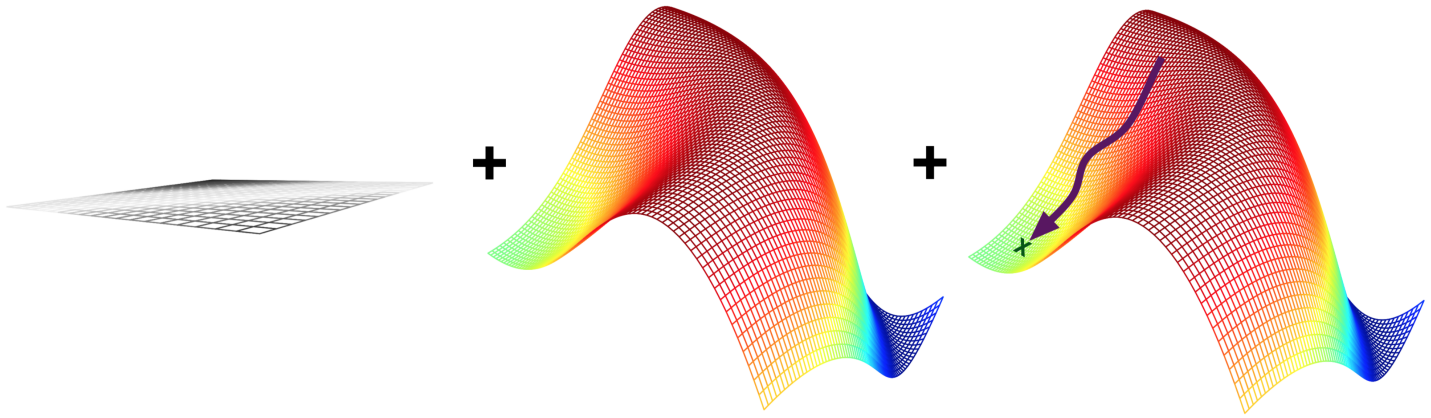
# Example: neural networks

- Representation: (layered) neural network
  - Each connection has a *weight* (a.k.a. model parameters)
  - Each node receives the weighted input values and emits a new value
- The *hypothesis space* consists of the set of all weights
- The architecture, type of neurons, etc. are fixed
  - We call these *hyper-parameters* (set by user, fixed during training)
  - They can also be learned (in an outer loop)



# Example: neural networks

- Representation: For illustration, consider the space of 2 model parameters
- Evaluation: A *loss function* computes, for each set of parameters, how good the predictions are
    - *Estimated* on a set of training data with the 'correct' predictions
    - We can't see the full surface, only evaluate specific sets of parameters
- Optimization: Find the optimal set of parameters
    - Usually a type of *search* in the hypothesis space
    - Given a few initial evaluations, predict which parameters may be better



# Generalization, Overfitting and Underfitting

- We *hope* that the model can *generalize* from the training data: make accurate predictions on unseen data.
- We can never be sure, only hope that we make the right assumptions.
    - We typically assume that new data will be similar to previous data
    - *Inductive bias*: assumptions that we put into the algorithm (everything except the training data itself)

## Example: Dating

| Nr | Day of Week | Type of Date | Weather | TV Tonight | Date? |
|---|---|---|---|---|---|
| 1 | Weekday | Dinner | Warm | Bad | No |
| 2 | Weekend | Club | Warm | Bad | Yes |
| 3 | Weekend | Club | Warm | Bad | Yes |
| 4 | Weekend | Club | Cold | Good | No |
| Now | Weekend | Club | Cold | Bad | ? |

- Can you find a simple rule that works? Is one better than others?
- What can we assume about the future? Nothing?
- What if there is noise / errors?
- What if there are factor you don't know about?

# Overfitting and Underfitting

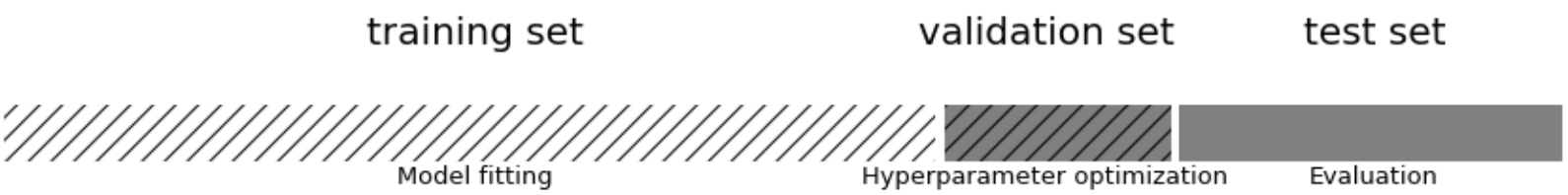
- It's easy to build a complex model that is 100% accurate on the training data, but very bad on new data
- Overfitting: building a model that is *too complex for the amount of data* that we have
    - You model peculiarities in your training data (noise, biases,...)
    - Solve by making model simpler (regularization), or getting more data
    - **Most algorithms have hyperparameters that allow regularization**
- Underfitting: building a model that is *too simple given the complexity of the data*
    - Use a more complex model
- There are techniques for detecting overfitting (e.g. bias-variance analysis). More about that later
- You can build *ensembles* of many models to overcome both underfitting and overfitting

- There is often a sweet spot that you need to find by optimizing the choice of algorithms and hyperparameters, or using more data.
- Example: regression using polynomial functions

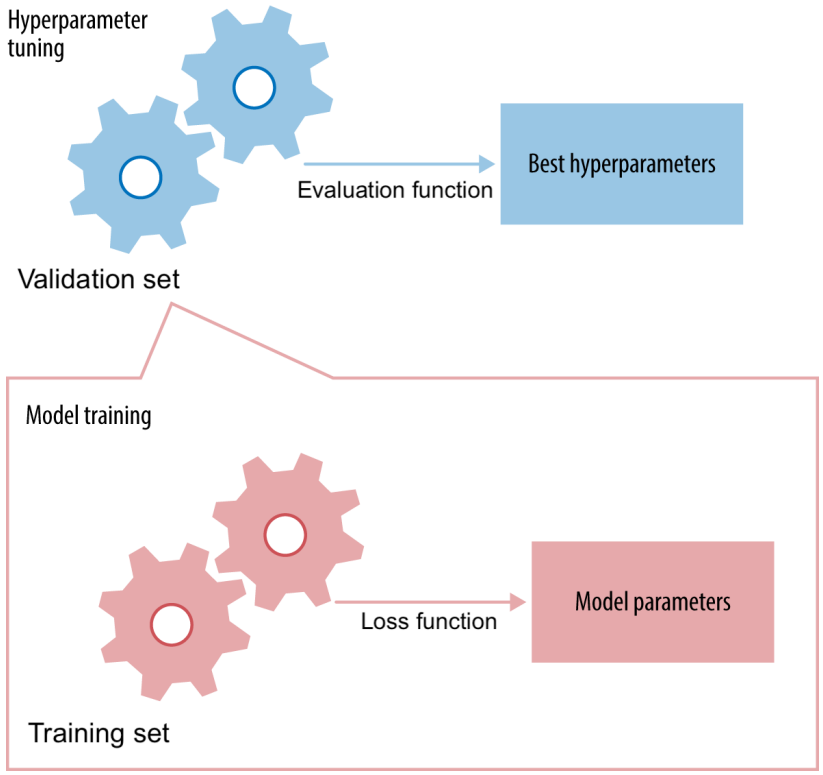
# Model selection

- Next to the (internal) loss function, we need an (external) evaluation function
    - Feedback signal: are we actually learning the right thing?
    - Are we under/overfitting?
    - More freely chosen to fit the application. Loss functions have constraints (e.g. differentiable)
    - Needed to choose between algorithms (or different hyper-parameter settings)

- Data needs to be split into *training* and *test* sets
    - Optimize model parameters on the training set, evaluate on independent test set
    - To optimize hyperparameters as well, set aside part of training set as a *validation* set



# Overview

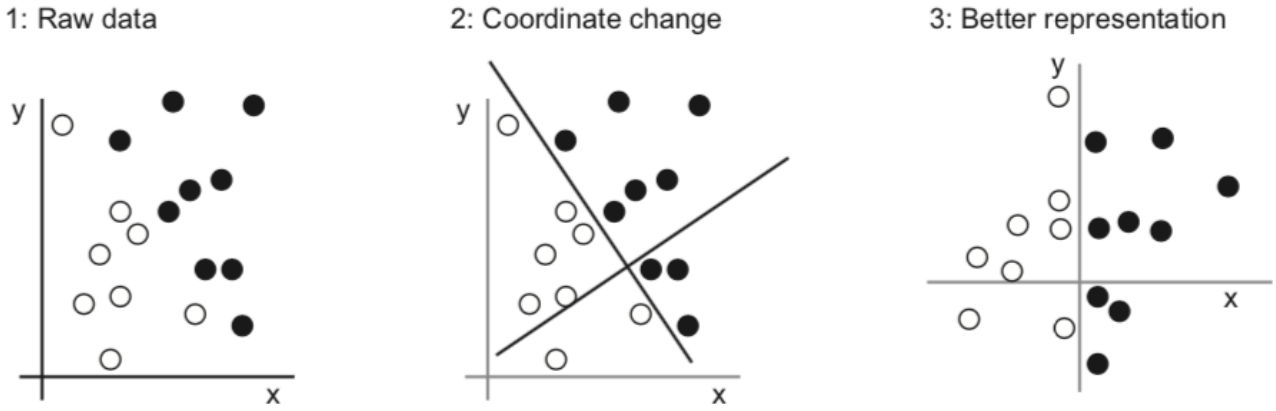

Hyperparameter tuning
Evaluation function
Best hyperparameters
Validation set
Model training
Loss function
Model parameters
Training set

# Only generalization counts!

- Never evaluate your final models on the training data, except for:
    - Tracking whether the optimizer converges (learning curves)
    - Detecting under/overfitting:
        - Low training and test score: underfitting
        - High training score, low test score: overfitting
- Always keep a completely independent test set
- Avoid data leakage:
    - Never optimize hyperparameter settings on the test data
    - Never choose preprocessing techniques based on the test data
- On small datasets, use multiple train-test splits to avoid bias
    - E.g. Use cross-validation (see later)

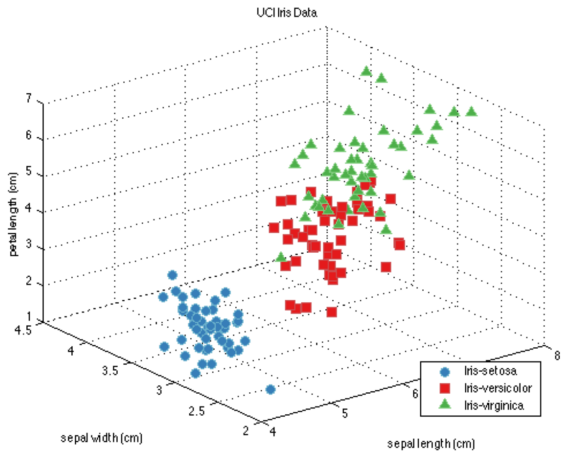
# Data (problem) representation

- Algorithm needs to correctly transform the inputs to the right outputs
- A lot depends on how we present the data to the algorithm
    - Transform the data to a more useful representation (a.k.a. *encoding* or *embedding*)
    - Can be done end-to-end (e.g. deep learning) or by first 'preprocessing' the data



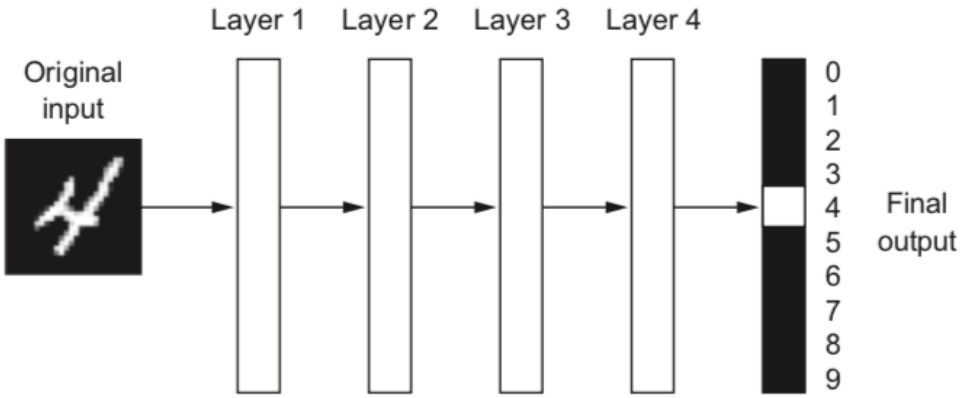
# Feature engineering

- Most machine learning techniques require humans to build a good representation of the data
    - Sometimes data is naturally structured (e.g. medical tests)
- Nothing beats domain knowledge (when available) to get a good representation
    - E.g. Iris data: leaf length/width separate the classes well
- Feature engineering is often necessary to get the best results
    - Feature selection, dimensionality reduction, scaling, ...
    - *Applied machine learning is basically feature engineering (Andrew Ng)*



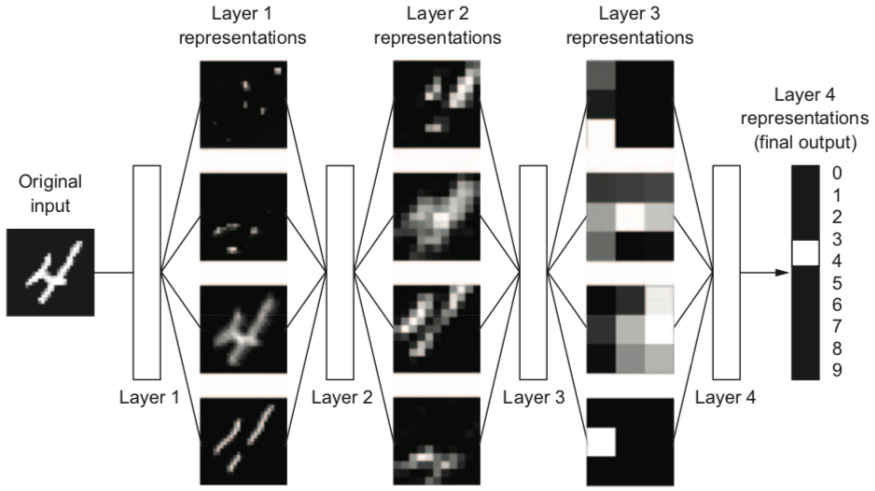
# Learning data transformations end-to-end

- For unstructured data (e.g. images, text), it's hard to extract good features
- Deep learning: learn your own representation (embedding) of the data
    - Through multiple layers of representation (e.g. layers of neurons)
    - Each layer transforms the data a bit, based on what reduces the error



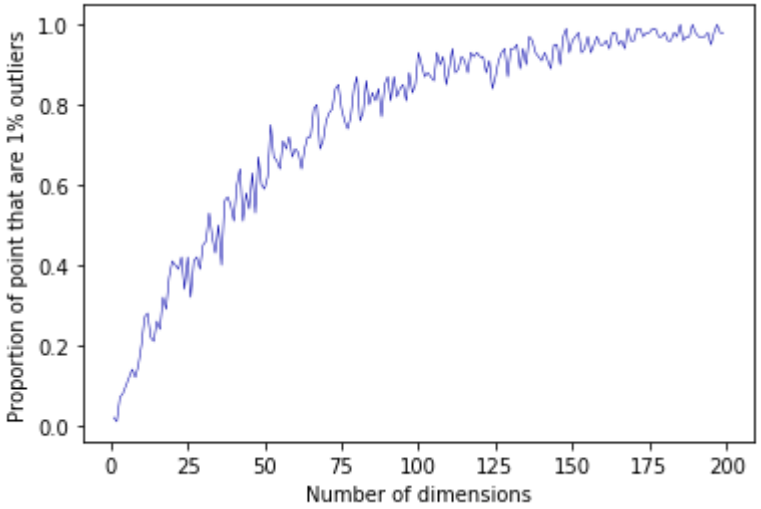
## Example: digit classification

- Input pixels go in, each layer transforms them to an increasingly informative representation for the given task
- Often less intuitive for humans



# Curse of dimensionality

- Intuition fails in high dimensions:
  - Randomly sample points in an n-dimensional space (e.g. a unit hypercube)
  - The more dimensions you have, the more sparse the space becomes
  - Distances between any two points will become almost identical
  - Almost all points become outliers at the edge of the space



**Practical consequences**

- For every dimension (feature) you add, you need exponentially more data to avoid sparseness
- Affects any algorithm that is based on distances (e.g. kNN, SVM, kernel-based methods, tree-based methods,...)
- Blessing of non-uniformity: on many applications, the data lives in a very small subspace
- You can drastically improve performance by selecting features or using lower-dimensional data representations

# "More data can beat a cleverer algorithm" (but you need both)

- More data reduces the chance of overfitting
- Less sparse data reduces the curse of dimensionality
- *Non-parametric* models: number of model parameters grows with the amount of data
    - Tree-based techniques, k-Nearest neighbors, SVM,...
    - They can learn any model given sufficient data (but can get stuck in local minima)
- *Parametric* (fixed size) models: fixed number of model parameters
    - Linear models, Neural networks,...
    - Can be given a huge number of parameters to benefit from more data
    - Deep learning models can have millions of weights, learn almost any function.
- The bottleneck is moving from data to compute/scalability

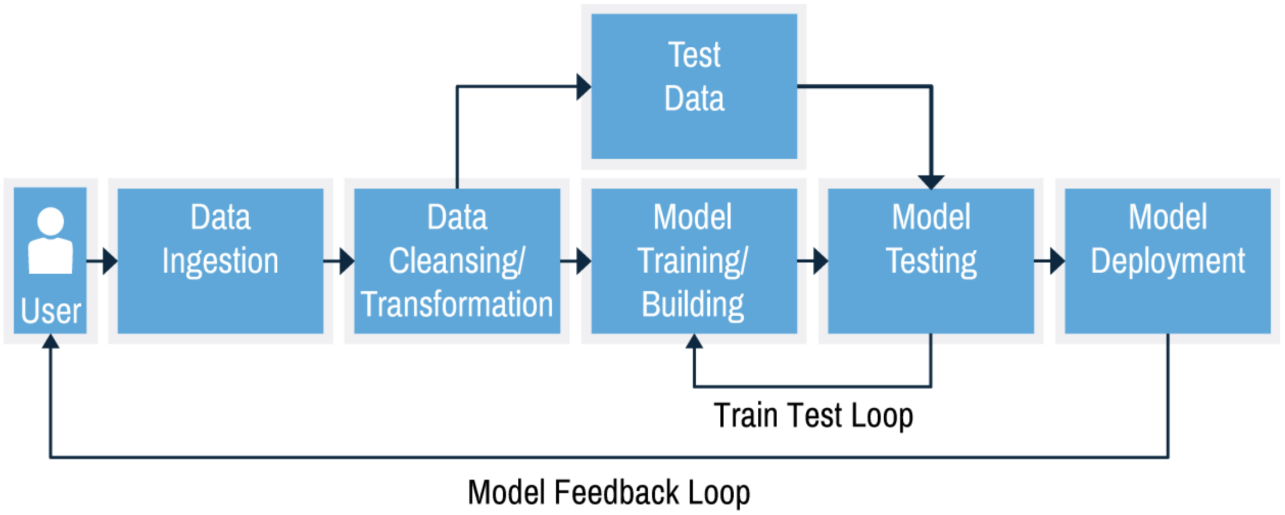
# Building machine learning systems

A typical machine learning system has multiple components:

- Preprocessing: Raw data is rarely ideal for learning (Lecture 4)
    - Feature scaling: bring values in same range
    - Encoding: make categorical features numeric
    - Discretization: make numeric features categorical
    - Label imbalance correction (e.g. downsampling)
    - Feature selection: remove uninteresting/correlated features
    - Dimensionality reduction can also make data easier to learn
    - Using pre-learned embeddings (e.g. word-to-vector, image-to-vector)

- Learning and evaluation (Lecture 3)
    - Every algorithm has its own biases
    - No single algorithm is always best
    - *Model selection* compares and selects the best models
        - Different algorithms, different hyperparameter settings
    - Split data in training, validation, and test sets
- Prediction
    - Final optimized model can be used for prediction
    - Expected performance is performance measured on *independent* test set

- Together they form a *workflow* of *pipeline*
- There exist machine learning methods to automatically build and tune these pipelines (Lecture 7)
- You need to optimize pipelines continuously
    - *Concept drift*: the phenomenon you are modelling can change over time
    - *Feedback*: your model's predictions may change future data



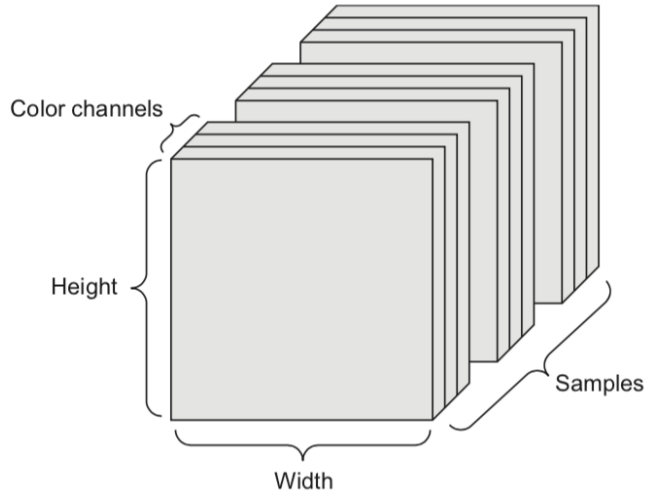
# The Mathematics behind Machine Learning

- We don't want you to run machine learning algorithms blindly, you need to understand what they do.
- To understand machine learning algorithms, it often helps to describe them mathematically.
- To avoid confusion, let's specify a precise notation

## Basic notation

- A *scalar* is a simple numeric value, denoted by italic letter: $x = 3.24$
- A *vector* is a 1D ordered array of $n$ scalars, denoted by bold letter: $\mathbf{x} = [3.24, 1.2]$
    - A vector can represent a *point* in an n-dimensional space, given a *basis*.
    - $x_i$ denotes the $i$th element of a vector, thus $x_0 = 3.24$.
        - Note: some other courses use $x^{(i)}$ notation
- A *set* is an *unordered* collection of unique elements, denote by caligraphic capital: $\mathcal{S} = \{3.24, 1.2\}$
- A *matrix* is a 2D array of scalars, denoted by bold capital: $\mathbf{X} = \begin{bmatrix} 3.24 & 1.2 \\ 2.24 & 0.2 \end{bmatrix}$
    - It can represent a set of points in an n-dimensional space, given a *basis*.
    - $\mathbf{X}_i$ denotes the $i$th *row* of the matrix
    - $\mathbf{X}_{i,j}$ denotes the *element* in the $i$th row, $j$th column, thus $\mathbf{X}_{0,1} = 2.24$
- The *standard basis* for a Euclidean space is the set of unit vectors
    - Data can also be represented in a non-standard basis (e.g. polynomials) if useful

- A *tensor* is an *k*-dimensional array of data, denoted by an italic capital: $T$
  - $k$ is also called the *order*, *degree*, or *rank*
  - $T_{i,j,k,...}$ denotes the element or sub-tensor in the corresponding position
  - A set of color images can be represented by:
    - a 4D tensor (sample x height x weight x color channel)
    - a 2D tensor (sample x flattened vector of pixel values)



## Basic operations

- Sums and products are denoted by capital Sigma and capital Pi:
$$\sum_{i=0}^{n} = x_0 + x_1 + \ldots + x_p \qquad \prod_{i=0}^{n} = x_0 \cdot x_1 \cdot \ldots \cdot x_p$$
- Operations on vectors are *element-wise*: e.g.
$\mathbf{x} + \mathbf{z} = [x_0 + z_0, x_1 + z_1, \ldots, x_p + z_p]$
- Dot product
$\mathbf{wx} = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{p} w_i \cdot x_i = w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_p \cdot x_p$
- Matrix product $\mathbf{Wx} = \begin{bmatrix} \mathbf{w_0} \cdot \mathbf{x} \\ \ldots \\ \mathbf{w_p} \cdot \mathbf{x} \end{bmatrix}$
- A function $f(x) = y$ relates an input element $x$ to an output $y$
    - It has a *local minimum* at $x = c$ if $f(x) \geq f(c)$ in interval $(c - \epsilon, c + \epsilon)$
    - It has a *global minimum* at $x = c$ if $f(x) \geq f(c)$ for any value for $x$
- A vector function consumes an input and produces a vector: $\mathbf{f}(\mathbf{x}) = \mathbf{y}$
- $\max_{x \in X} f(x)$ returns the highest value f(x) for any x
- $\operatorname{argmax}_{c \in C} f(x)$ returns the element c that maximizes f(c)

# Gradients

- A *derivative* $f'$ of a function $f$ describes how fast $f$ grows or decreases
- The process of finding a derivative is called differentiation
    - Derivatives for basic functions are known
    - For non-basic functions we use the *chain rule*:
      $F(x) = f(g(x)) \rightarrow F'(x) = f'(g(x))g'(x)$
- A function is *differentiable* if it has a derivate in any point of it's domain
    - It's *continuously differentiable* if $f'$ is itself a function
    - It's *smooth* if $f', f'', f''', \ldots$ all exist
- A *gradient* $\nabla f$ is the derivate of a function in multiple dimensions
    - It is a vector of *partial derivatives*: $\nabla f = \left[\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \ldots\right]$
    - E.g. $f = 2x_0 + 3x_1^2 - \sin(x_2) \rightarrow \nabla f = [2, 6x_1, -cos(x_2)]$