Machine Learning Tutorial

Prof Ing. Gianluca Valentino

Department of Communications and Computer Engineering

University of Malta

L-Università ta' Malta

ICALEPCS CAPE TOWN SOUTH AFRICA 2023

In the next 2 hours..

- ML 101: linear & logistic regression
- Different learning paradigms and tasks
- Neural Networks
- Clustering & Anomaly detection
- Advanced topics: CNNs and RL

Preliminary info before we dive in

- This tutorial only assumes:
  1. that you have some programming knowledge (ideally Python).
  2. that you have some data modeling experience (e.g. fitting a line to a curve).

- This tutorial will not make you a ML expert.. but at least you will:
  1. Grasp the **basic concepts** to be able to start learning more.
  2. Learn the importance of **looking under the hood**.
  3. Understand **which problem** would require **which technique/model.**
  4. Familiarize yourself with **commonly used Python libraries** for ML.

Outline

- **ML 101: linear & logistic regression**
- Different learning paradigms and tasks
- Neural Networks
- Clustering & Anomaly detection
- Advanced topics: CNNs and RL

ML 101: Linear Regression

- **Regression analysis:** a statistical process for estimating the relationship between variables

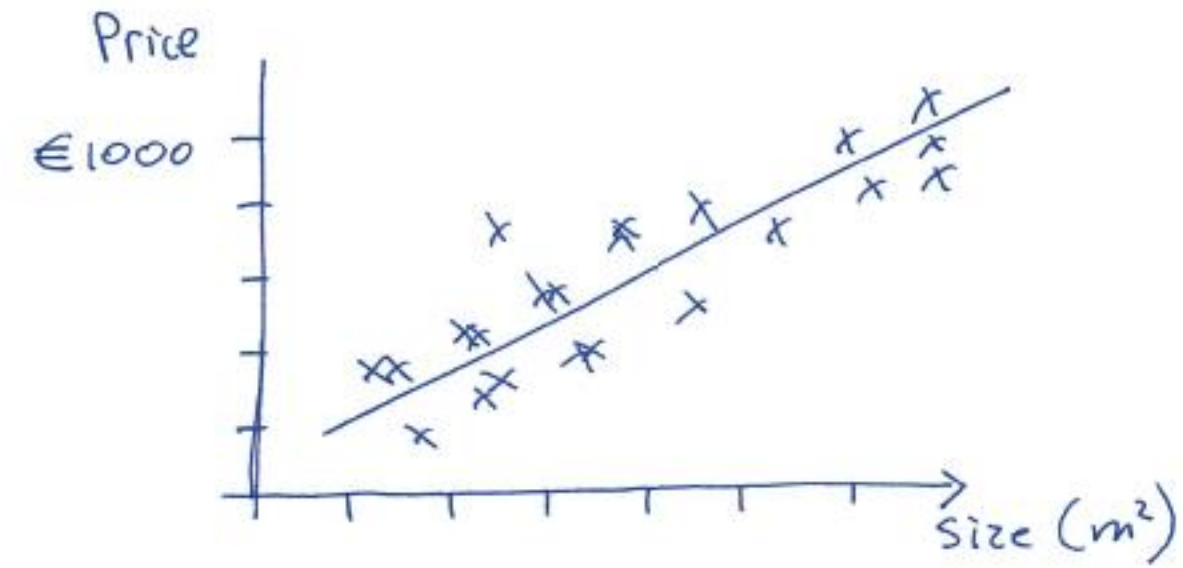
- Any regression model involves the following:

  - The independent variables X (known)
  - The dependent variable Y (known)
  - The vector of parameters θ (unknown)

where $Y \approx f(X, \theta)$

Linear Regression: example

- Consider apartment prices in Cape Town as a function of size.



- We would like to build a <u>model</u> that predicts <u>price</u> given a certain <u>size</u>.

- This is a case of *supervised learning*

Linear Regression: example

- Formally, we need to build a *dataset* (e.g. from estate agents)

- In this particular case, it is known as a *labelled* dataset.

- Notation:
  - m = # training examples
  - X = input variables/features
  - Y = output/target variables
  - (X,Y) = one training example

- We have *m* training examples. So training set is the matrix:

$[(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(i)}, y^{(i)}), \ldots (x^{(m)}, y^{(m)})]$

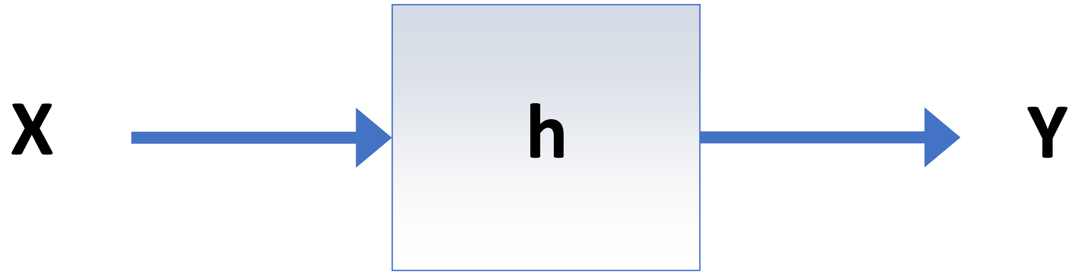
where i refers to the i^{th} training example

The hypothesis

- In general, we want to *discover* a model or hypothesis

- So in supervised learning, we:
  - start from a training set
  - learn a model which has a certain structure and parameters from the training set

- We need to define the model ourselves (e.g. a 2nd order polynomial).



Back to our house prices example..

1. Select a model (structure + parameters)

   - We can do this manually using visualization
   - We see a linear relationship between price and area
   - So the structure is that of a linear function with one variable:

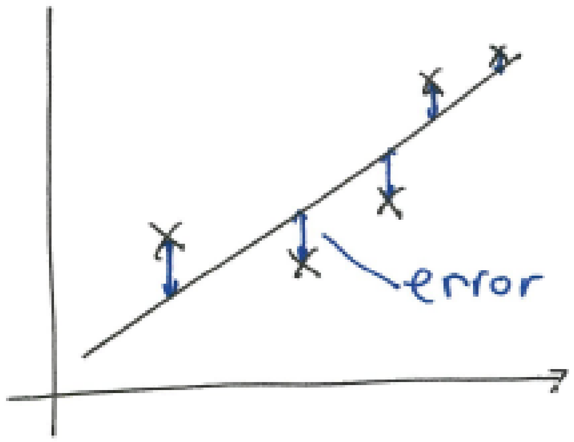
$$h_\theta(x) = \theta_0 + \theta_1 x$$

   - This is known as linear regression with one variable (or *univariate* linear regression)

Back to our house prices example..

2. The next step is to <u>learn</u> the model parameters

- We notice visually that the best fit is obtained when the Euclidean distance between each point and the line is *minimized*



$$J(\theta) = \min_{\theta} \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Cost Function

We want this term to be small ("squared error")

- We can define a <u>cost function</u> which minimizes the <u>error</u> between our predicted value $h_\theta(x)$ and our actual output y.

Summary so far..

- Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

- Parameters: θ_0, θ_1

- Cost function: $J(\theta) = \min_{\theta} \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

- Goal: to find values θ_0, θ_1 which minimize $J(\theta_0, \theta_1)$

Gradient Descent

- An algorithm for *iteratively* finding the minimum of a function

- A function is at its minimum when its gradient (found through differentiation) = 0

1. Start with a random [θ_0, θ_1]

2. Keep changing [θ_0, θ_1] in small steps to reduce $J(\theta)$ until a minimum is found

Gradient Descent

- Formally, we write:

REPEAT until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

}

where:

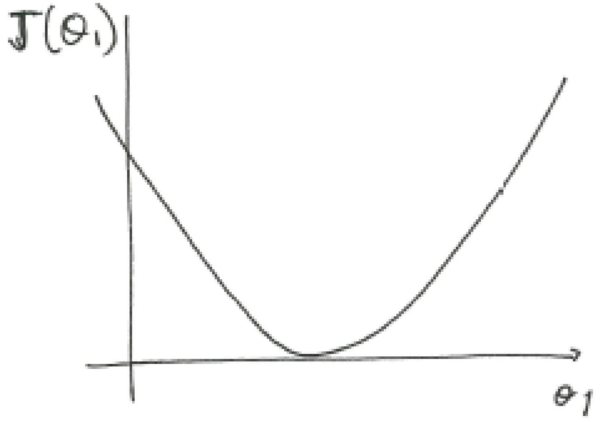
- α = learning rate (step size) **– a hyperparameter**
- $\frac{\partial}{\partial \theta_j}$ = partial derivative of J(θ) $= \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

- All θ_j are updated simultaneously

Gradient Descent

- Consider $h_\theta(x) = \theta_1 x$.

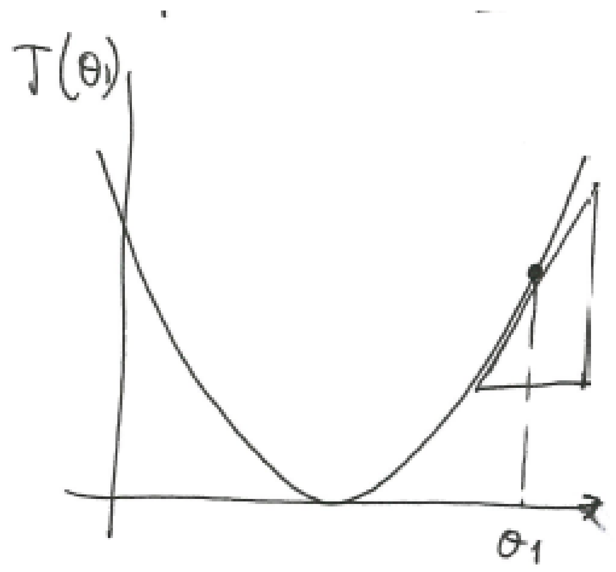
- We know that $J(\theta_1)$ looks like:



- Update equation is: $$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

Gradient Descent

- Suppose we start at:



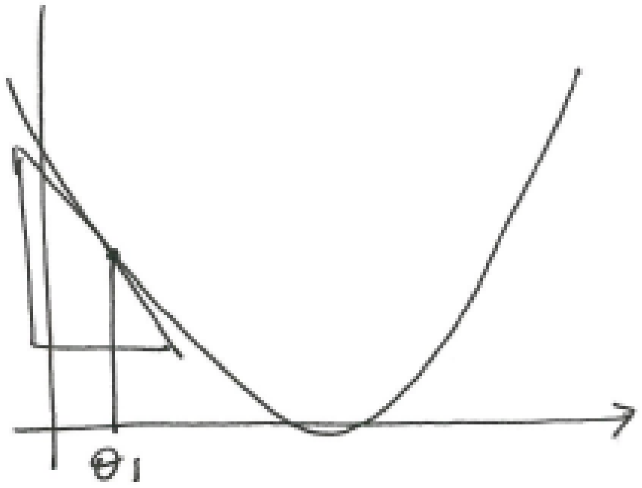
- Slope is positive here
- We want to move *downwards* so that $J(\theta_1)$ decreases
- We must *decrease* θ_1
- Therefore, the update equation must be:

$$\theta_1 := \theta_1 - \alpha \times (\text{positive number})$$

- θ_1 decreases as we want it to

Gradient Descent

- Instead, suppose we start at:



- Slope is negative here
- We want to move *downwards* so that $J(\theta_1)$ decreases
- We must *increase* θ_1
- Therefore, the update equation must be:

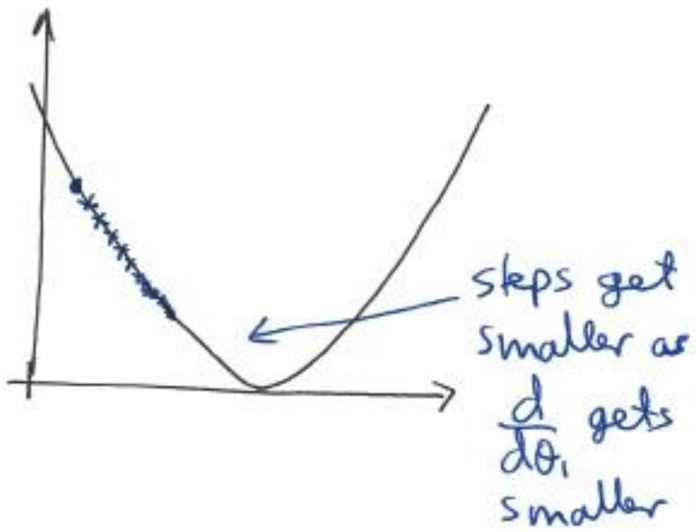
$$\theta_1 := \theta_1 - \alpha \times (\text{negative number})$$

- θ_1 increases as we want it to

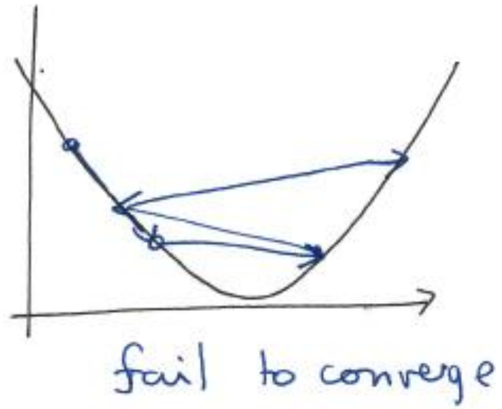
Selection of α

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

- If α is too small, then algorithm is slow

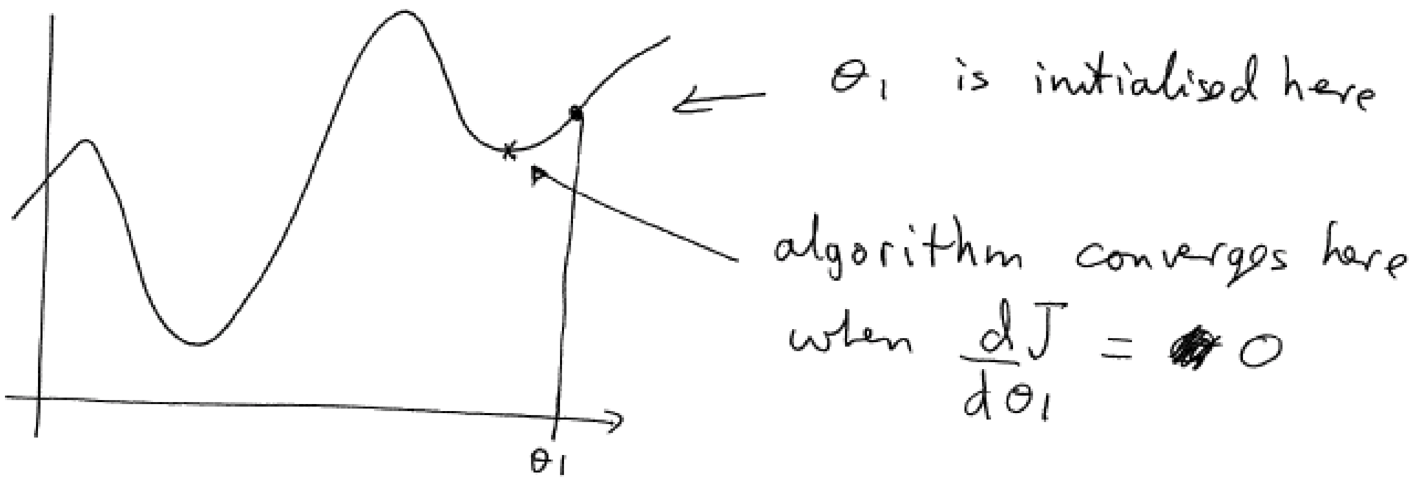


- If α is too large, then the algorithm could overshoot the minimum and fail to converge



Local vs global minima

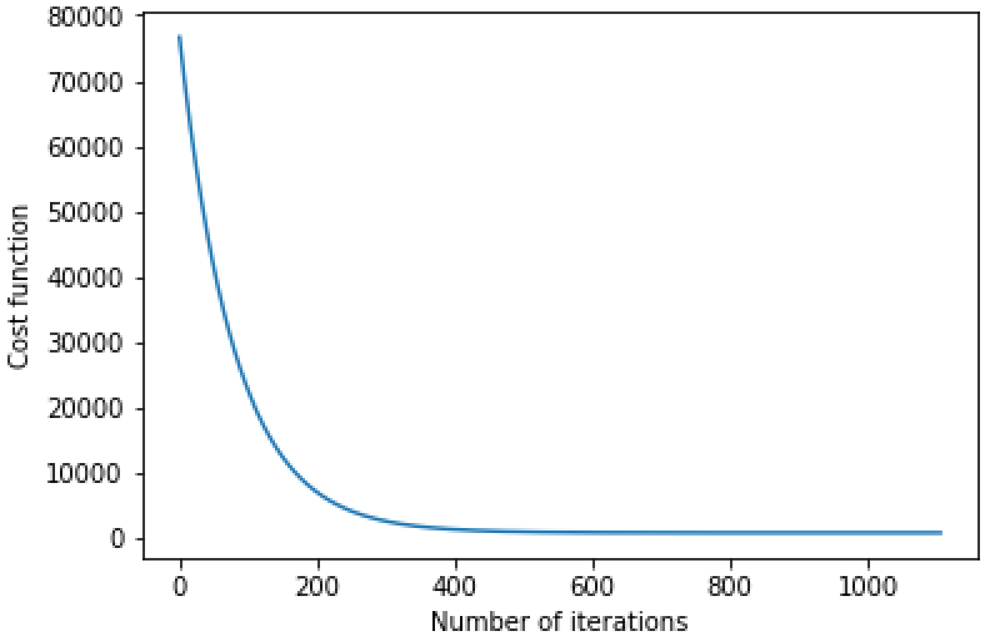
- A cost function may have more than one minimum



- In the case of the house price model **and all linear models**, J(θ) is a *convex* function, so there is only one minimum.

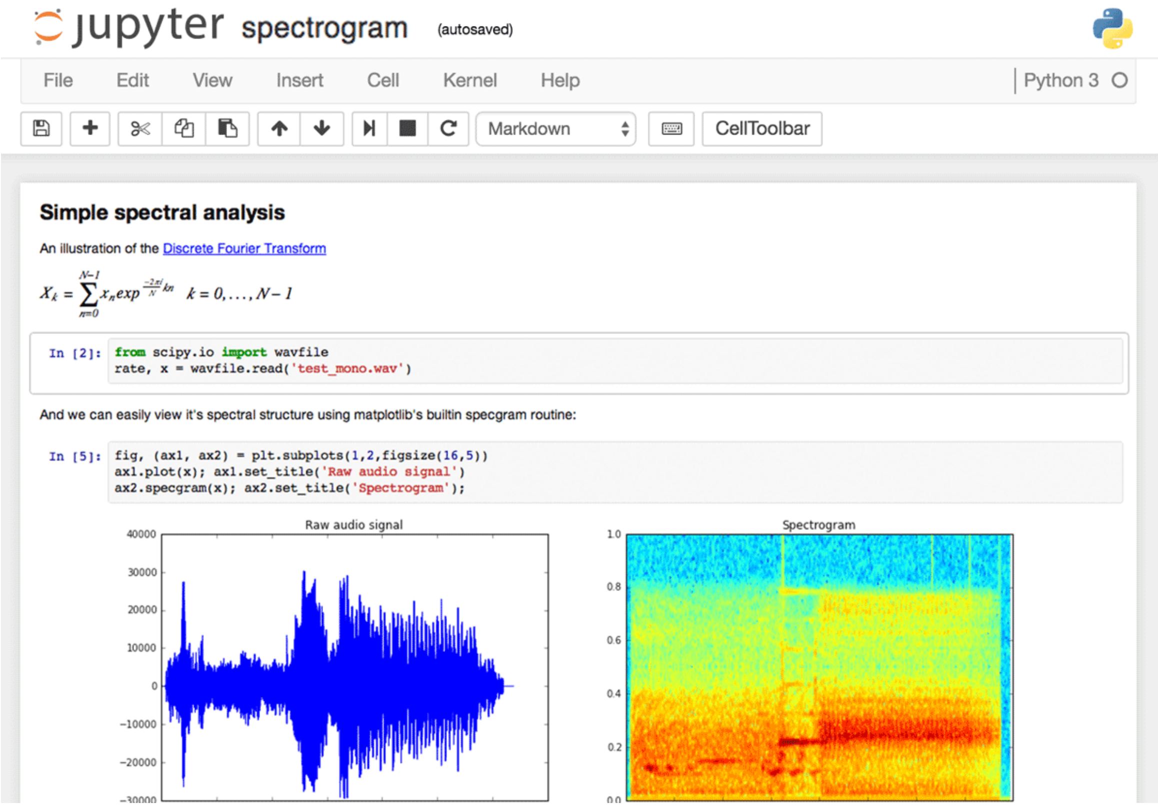
Performance evaluation

- There are several metrics which can be used when predicting **continuous** variables:
  - Mean square error: $\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$
  - Mean absolute error: $\frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$
  - R^2: $1 - \frac{\text{sum squared regression (SSR)}}{\text{total sum of squares (SST)}} = 1 - \frac{\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}{\sum_{i=1}^{n}|Y_i - \bar{Y}_i|^2}$
- We can calculate these metrics on both the **training set** (e.g. 80% of total data) and the unseen **testing set** (e.g. 20%)
- We can also observe the convergence of the model from the cost function vs # iterations



Jupyter notebooks

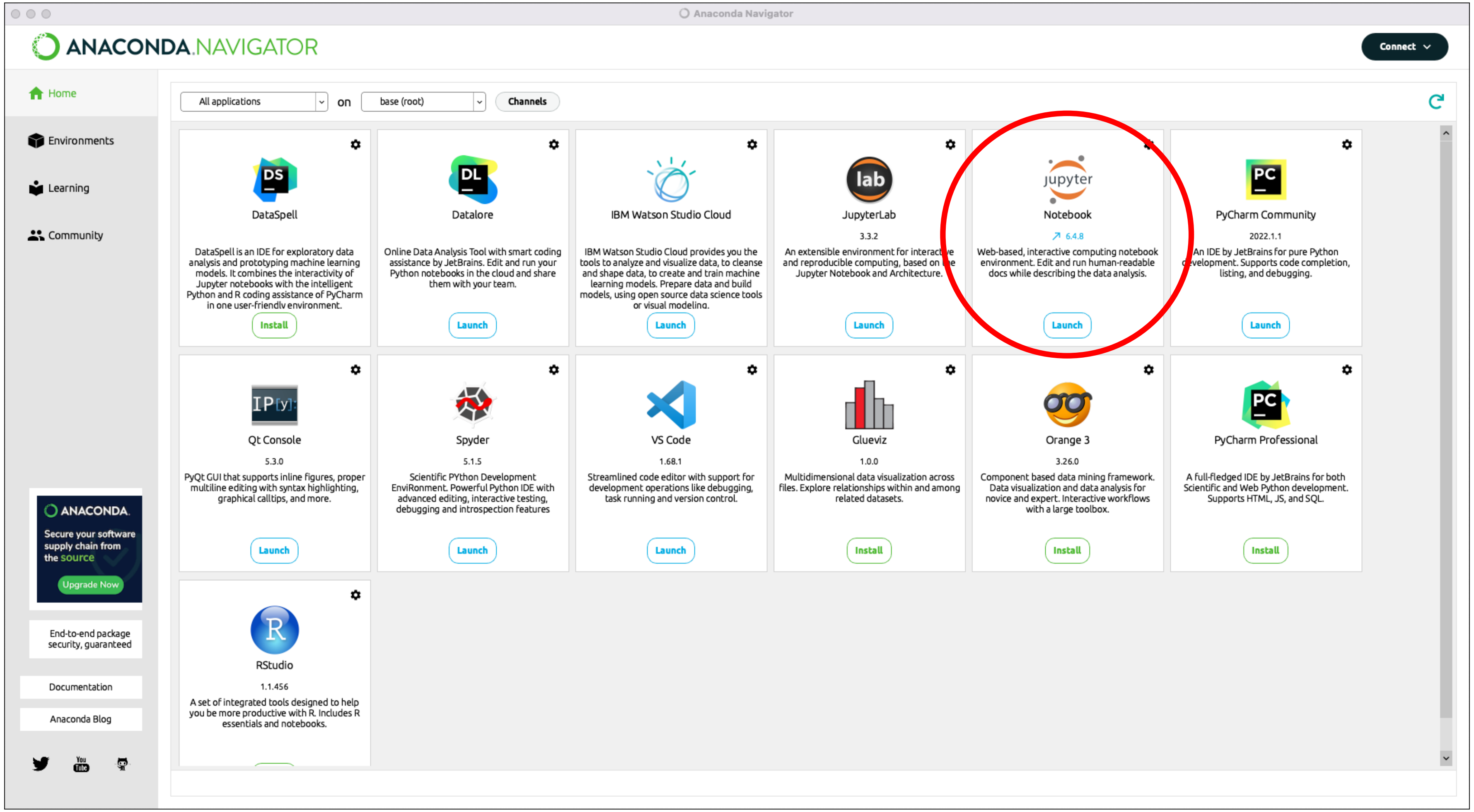
- Jupyter notebooks are a browser-based interactive development environment.



- There many possible setups, including launching from a terminal in a Python virtual environment or else using a GUI such as Anaconda (recommended for beginners)

Jupyter notebooks

- https://www.anaconda.com/download



Jupyter notebook

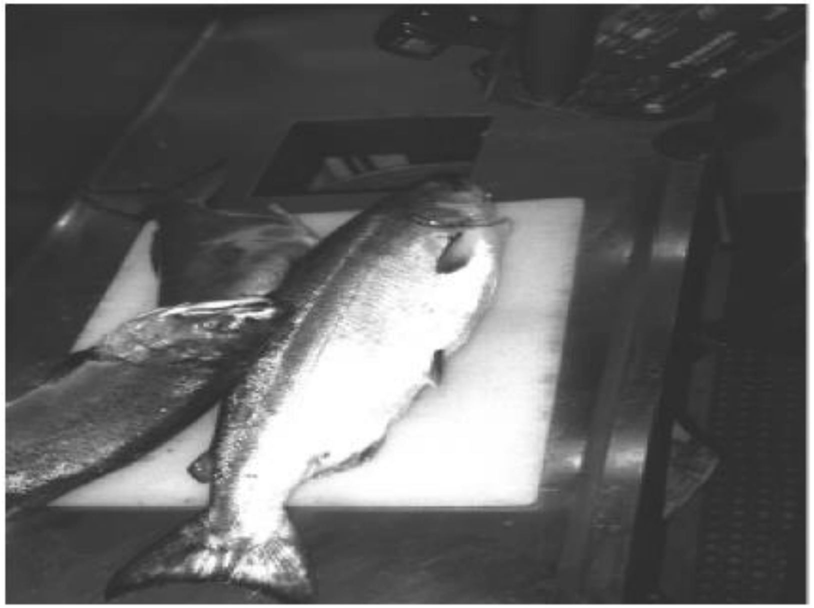
- Linear regression

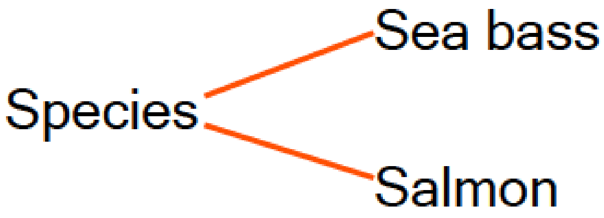
Linear Regression: summary

- **Terminology:** hypothesis, weights, hyperparameters, training/testing set,
- **Training** via an iterative process (gradient descent)
- We have seen the difference between
  - parameters/weights (e.g. theta) which are learnt during training
  - hyperparameters (e.g. alpha) which need to be set in advance
- We have seen how to **evaluate performance**

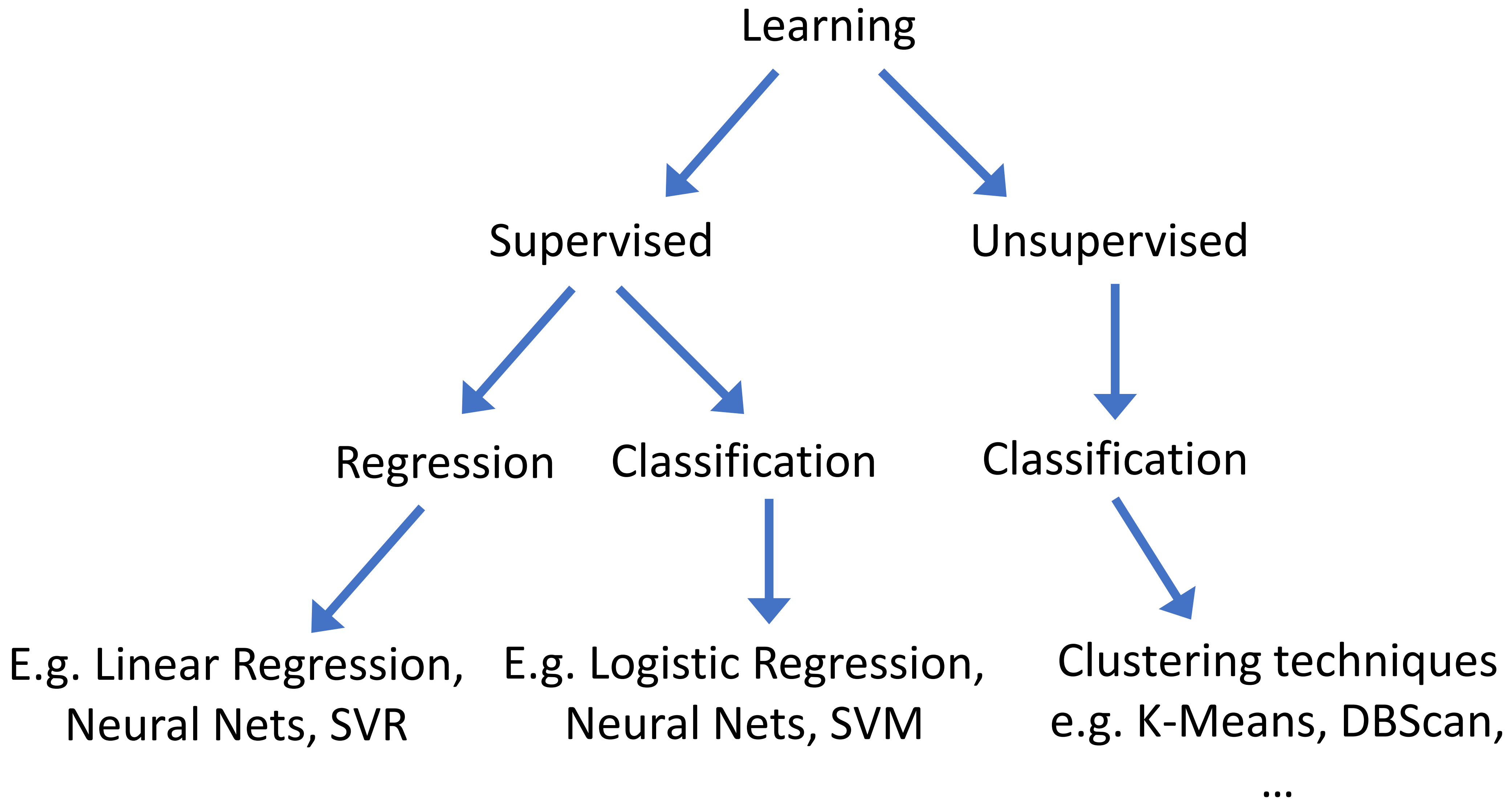
Introduction to Classification

- Consider a simple example:
  - "Sorting incoming fish on a conveyor belt according to species using optical sensors"

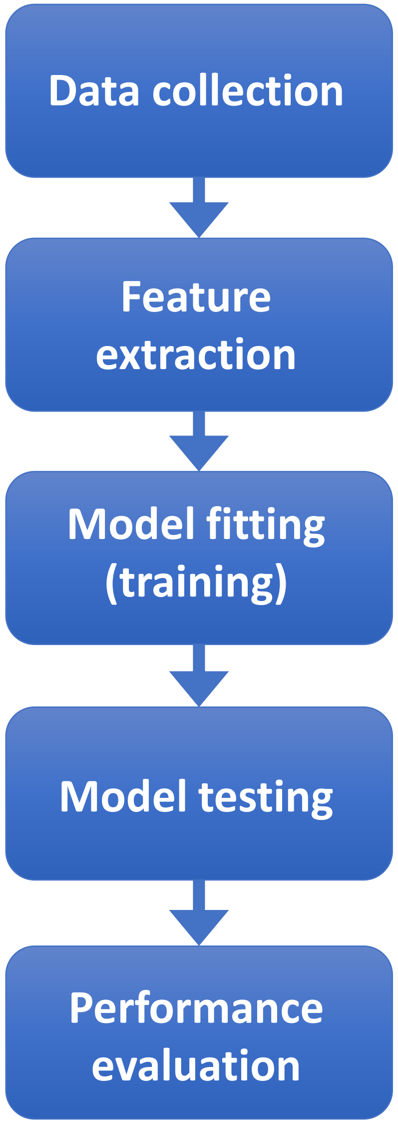




Machine Learning



Classification

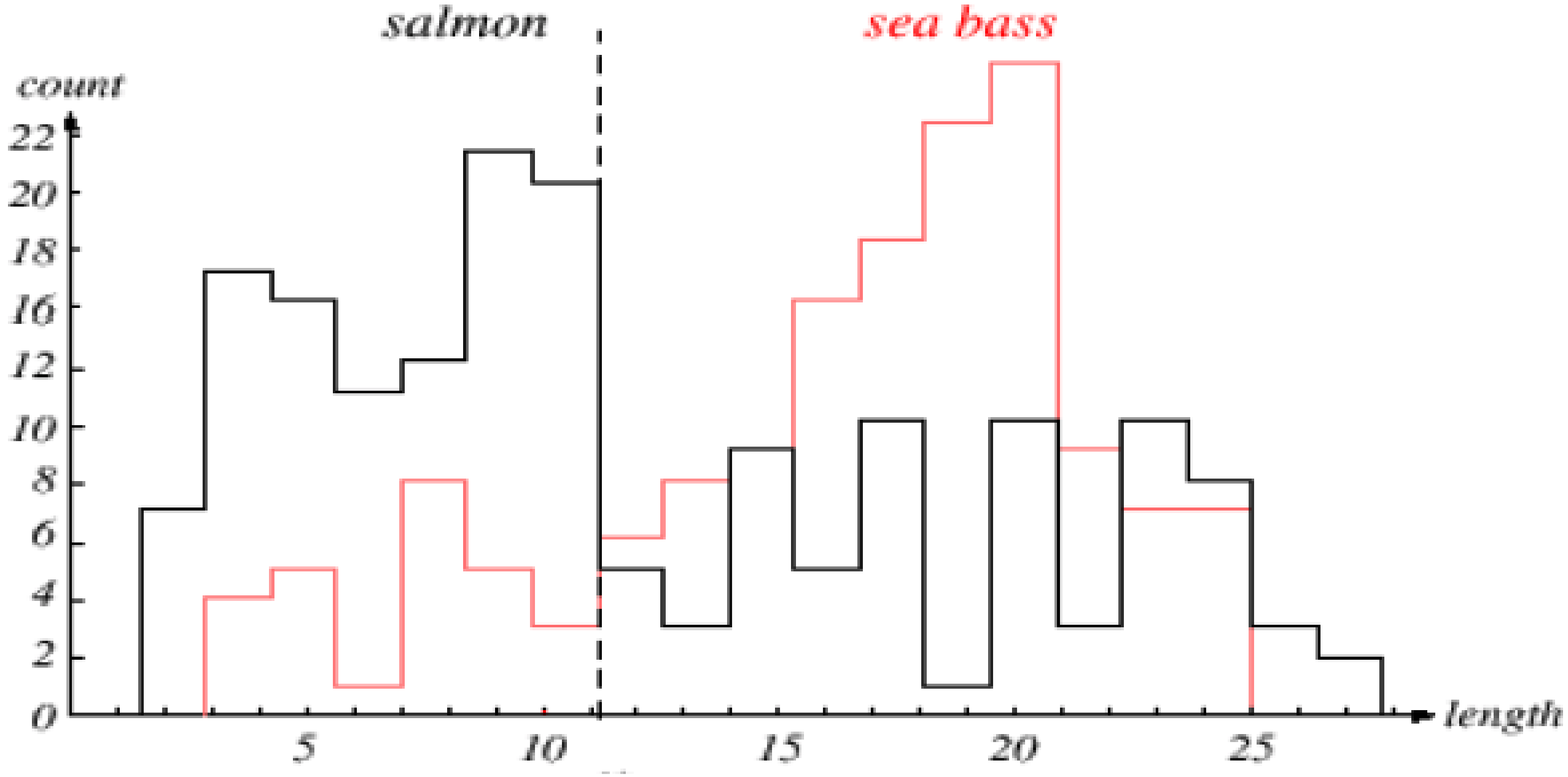


Feature extraction

- We have to think of features which could allow us to *discriminate* between salmon and sea bass
  - Length
  - Weight
  - Width
  - Number and shape of fins

Classification

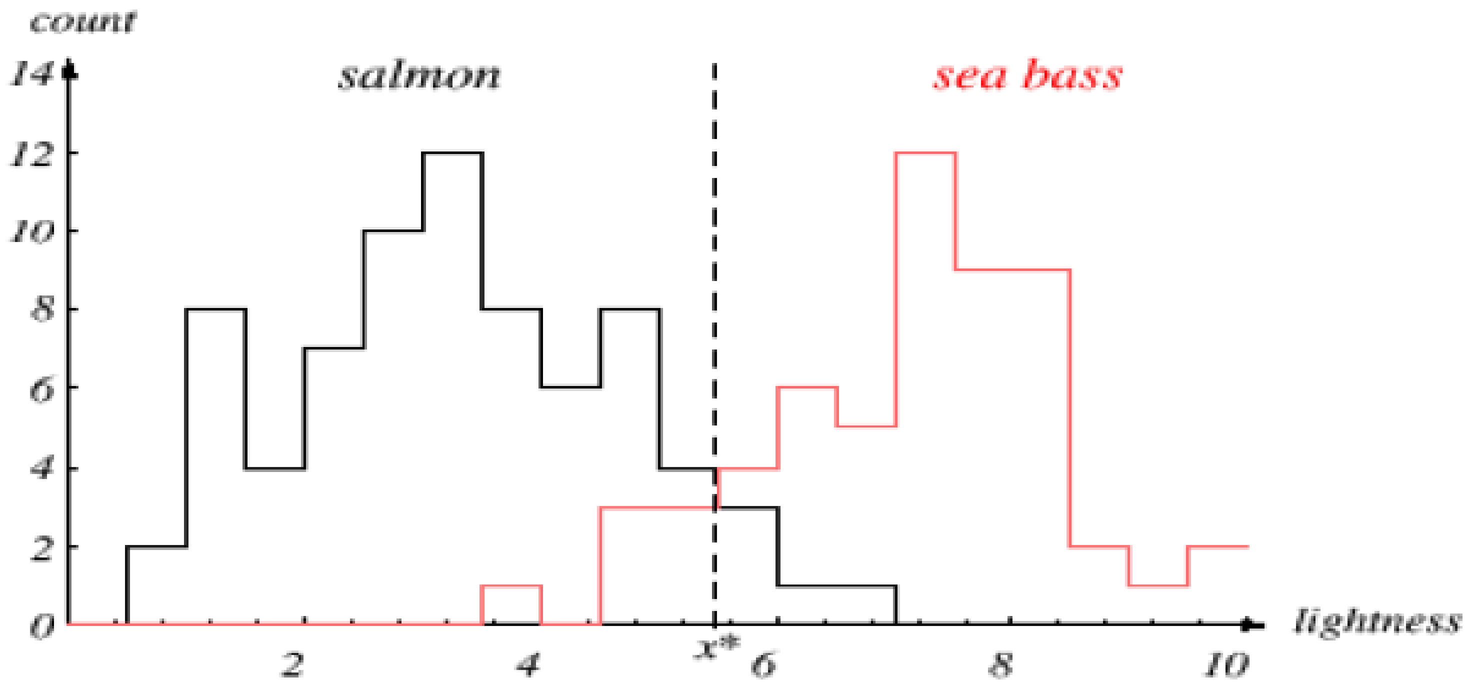
- Suppose we consider the length of the fish as a possible feature for discrimination



Preliminary results

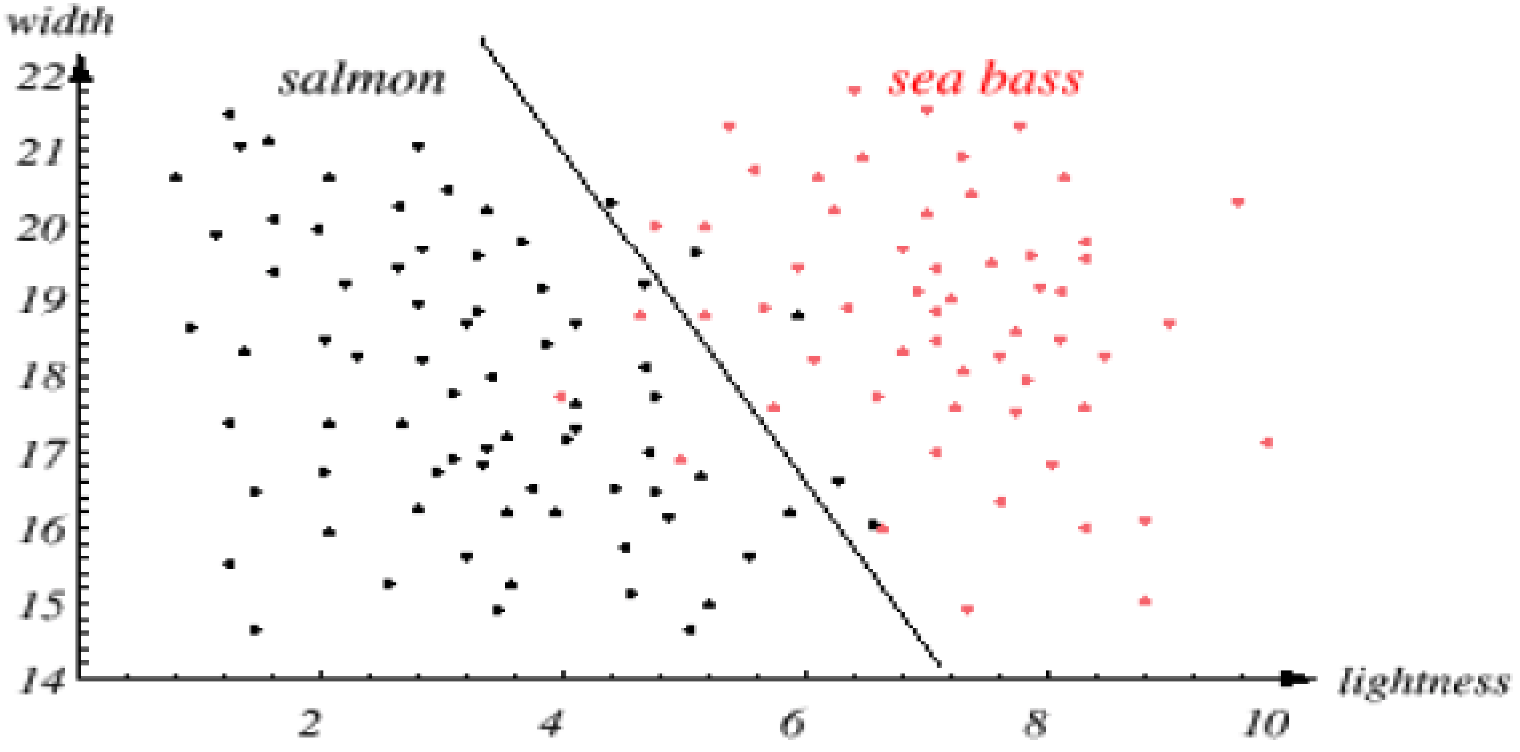
- We observe that the <u>length</u> on its own is a poor feature
  - About 20% misclassification rate

- Suppose we now select the <u>weight</u> as a possible feature



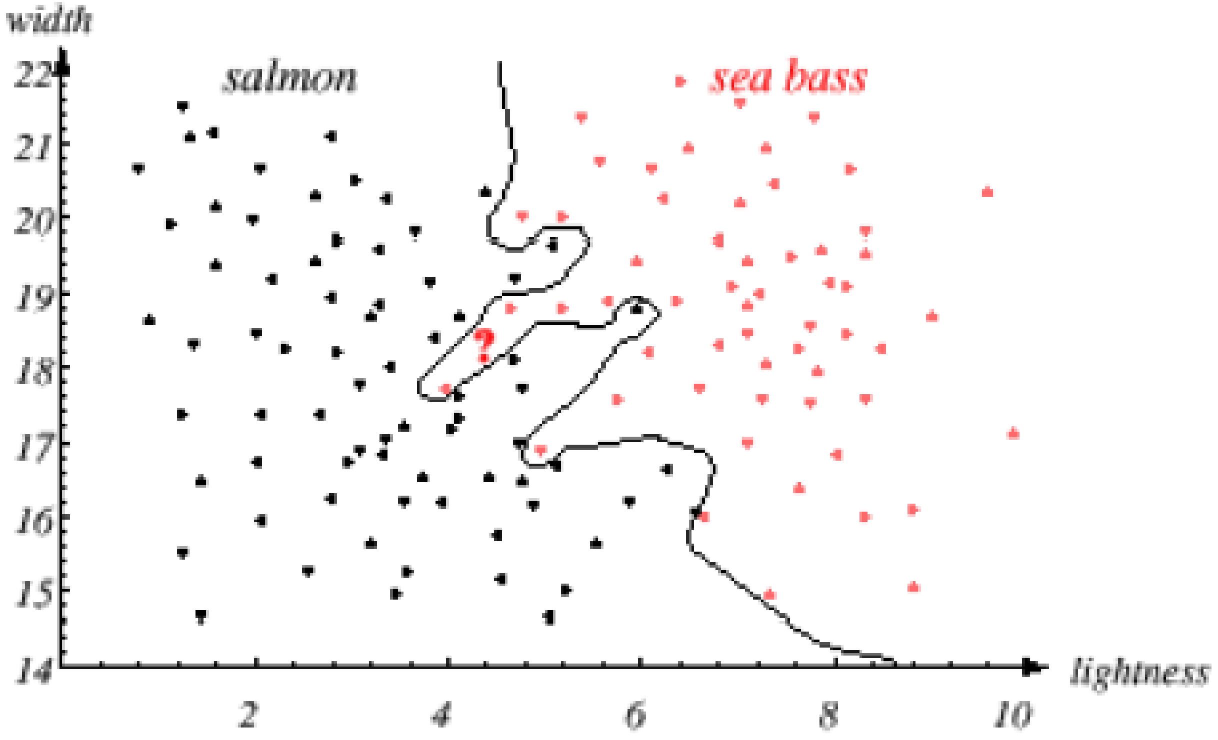
An improved classifier

- If we now combine the width and weight features:



Overfitting

- Naively, the best decision boundary would be the one below:



- However, this means that the model will not perform well for new data (therefore it does not generalize well)

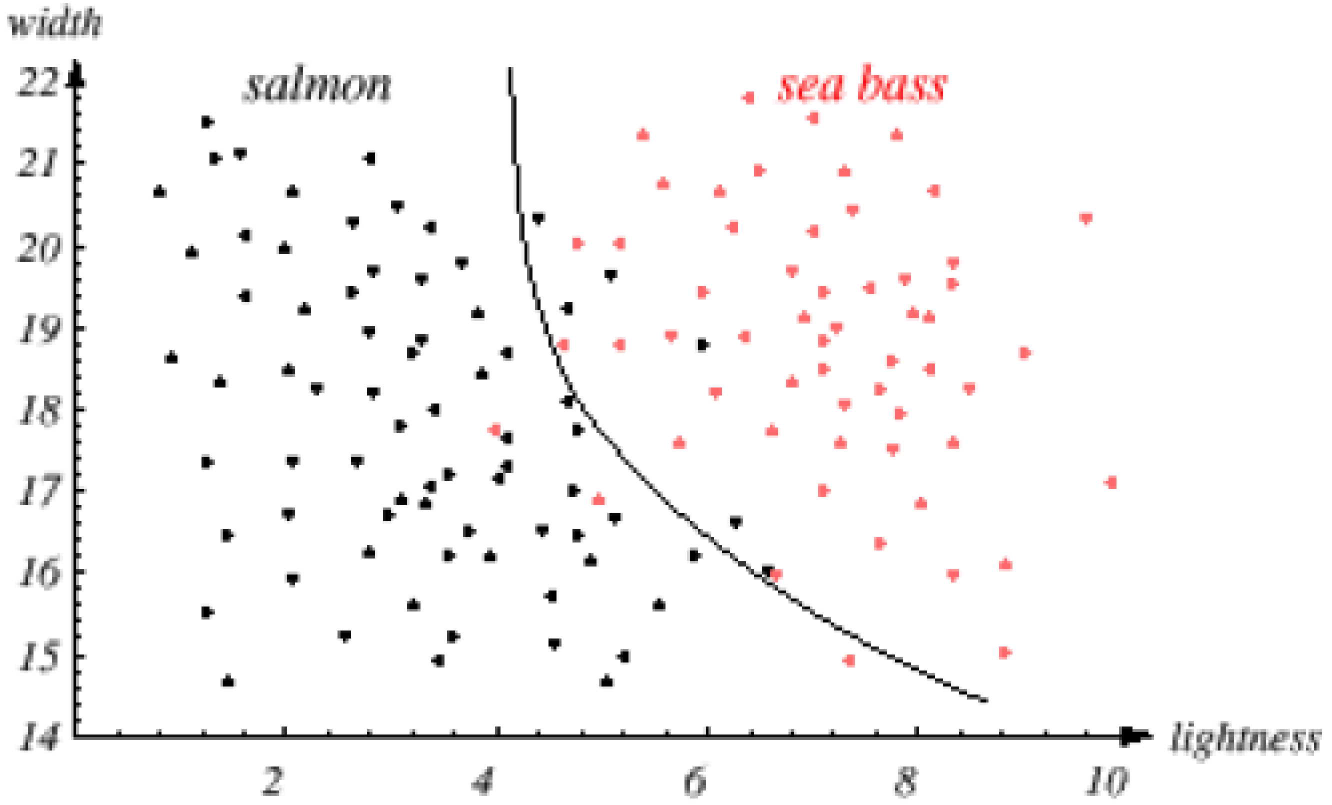
The classification problem

- Underfitting:
  - model not detailed enough
  - Bad performance on training and test datasets

- Overfitting:
  - Model too detailed and computationally expensive
  - Excellent performance on training set, bad performance on test set

An even better decision boundary

- A 2D polynomial might give the best fit and tradeoff:



Reminder: Linear Regression

- **Regression analysis:** a statistical process for estimating the relationship between variables

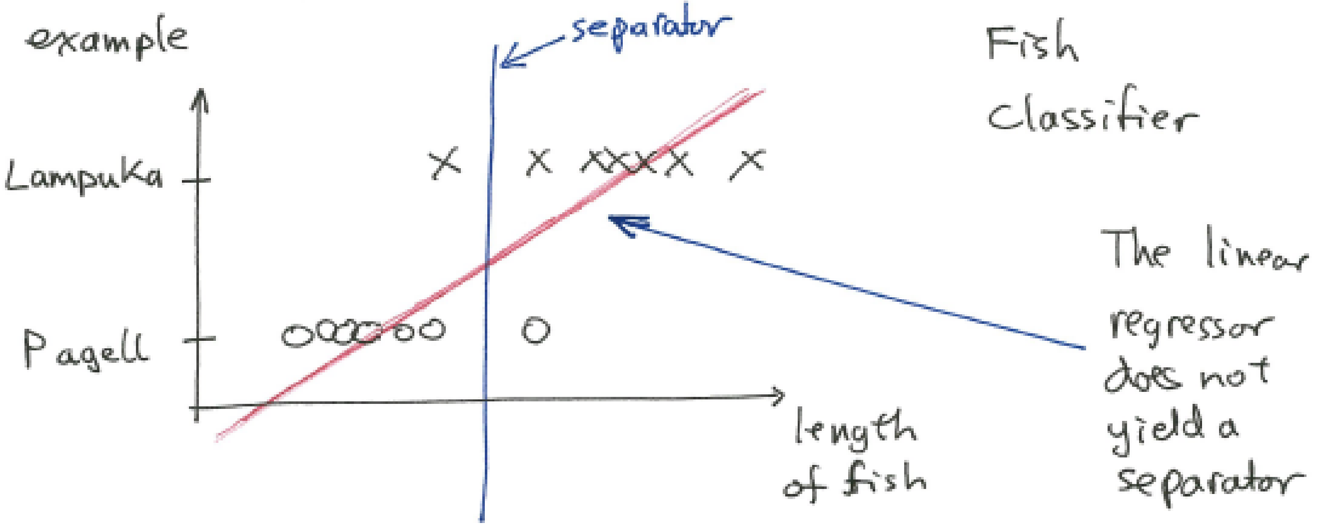
- Any regression model involves the following:

  - The independent variables X (known)
  - The dependent variable Y (known)
  - The vector of parameters θ (unknown)

where $Y \approx f(X, \theta)$

Linear regression

- Linear regressor does not work for classification:



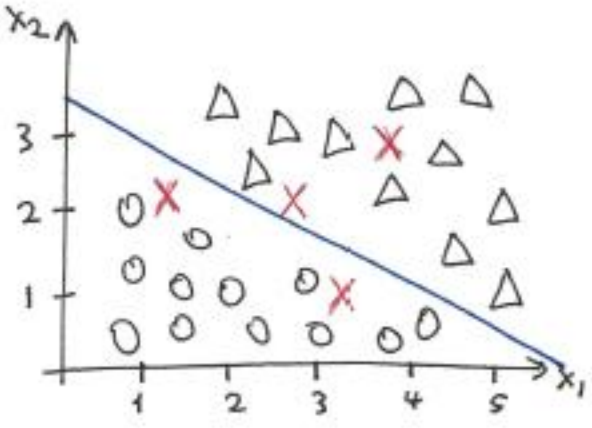
- This is a single-input **binary** class problem.

- In classification, we need a separator or a decision boundary which splits the space into regions.

Intuitive derivation of logistic regression model

- Consider a two-input binary class problem.

- Suppose we plot our data:



- Where:
  - x1, x2 are the inputs
  - O and Δ are the classes
  - X are the new instances

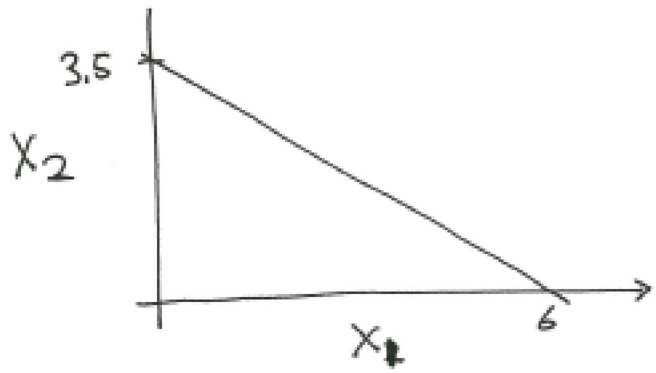
Intuitive derivation of logistic regression model

- Thanks to the separator:
  - new instances which fall above the line will be classified as Δ
  - new instances below the line will be classified as O.

- We need to automatically find a separator such as the one plotted.

Intuitive derivation of logistic regression model

- Suppose that we manage to come up with a valid equation for the separator visually



- The equation of the line is y = mx + c, and suppose that we measure m = -7/12 and c = +7/2

- So we have $X_2 = 7/2 – 7/12\ X_1$
- Or $7\ X_1 + 12\ X_2 – 42 = 0$

Intuitive derivation of logistic regression model

- If we choose points **on** the line, the equality holds.

- E.g. $\mathbf{X_1 = 3, X_2 = 7/4;}$ 7*3 + 12*7/4 – 42 = 0

- Now let $\mathbf{X_1 = 4, X_2 = 3;}$ 28 + 36 – 42 = **22 > 0**

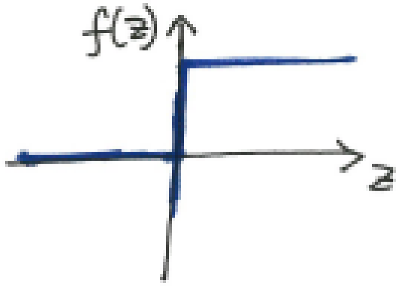
- And let $\mathbf{X_1 = 1, X_2 = 5/2;}$ 7 + 30 – 42 = **-5 < 0**

- So points below the line will give us –ve values, while points above the line give +ve values.

Intuitive derivation of logistic regression model

- So we could write our model as:
  - $h_\theta(x) = f(\theta_0 + \theta_1 X_1 + \theta_2 X_2)$

- We want our output to be either 0 or 1 (i.e. either the input belongs to one class, or else to the other):

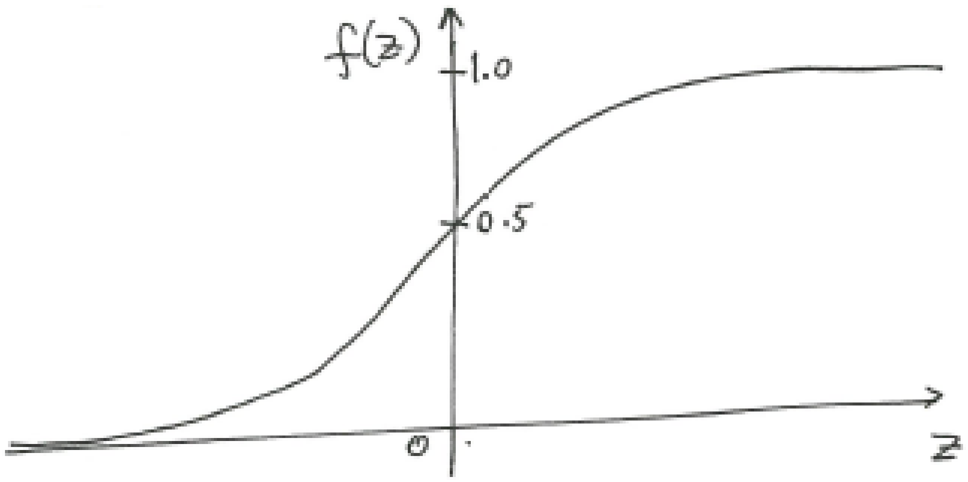


- However this is a non-differentiable, discontinuous function

- We like differentiable functions as it allows us to minimize the cost function using gradient descent ☺

Intuitive derivation of logistic regression model

- We would therefore prefer to use another function, such as the **sigmoid** or **logistic** function.



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

$$z \geq 0, \quad g(z) \geq 0.5$$

$$z < 0, \quad g(z) < 0.5$$

Cost function

- Inspired from the regression cost function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} log(h_\theta(x^{(i)})) + (1 - y^{(i)}) log(1 - h_\theta(x^{(i)}))$$

- <u>Explanation</u>:
  - By definition of logistic function, $h_\theta(x)$ values vary from 0 to 1
  - y is either 0 or 1
  - So:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} log(h_\theta(x^{(i)}))$$, if y = 1

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} log(1 - h_\theta(x^{(i)}))$$, if y = 0

Minimizing the cost function

- We use gradient descent as for linear regression
- We note that the partial differentiation of the cost function for θ_j is the same as for linear regression (!)

Update equation: $$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

Where: $$\frac{\partial}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

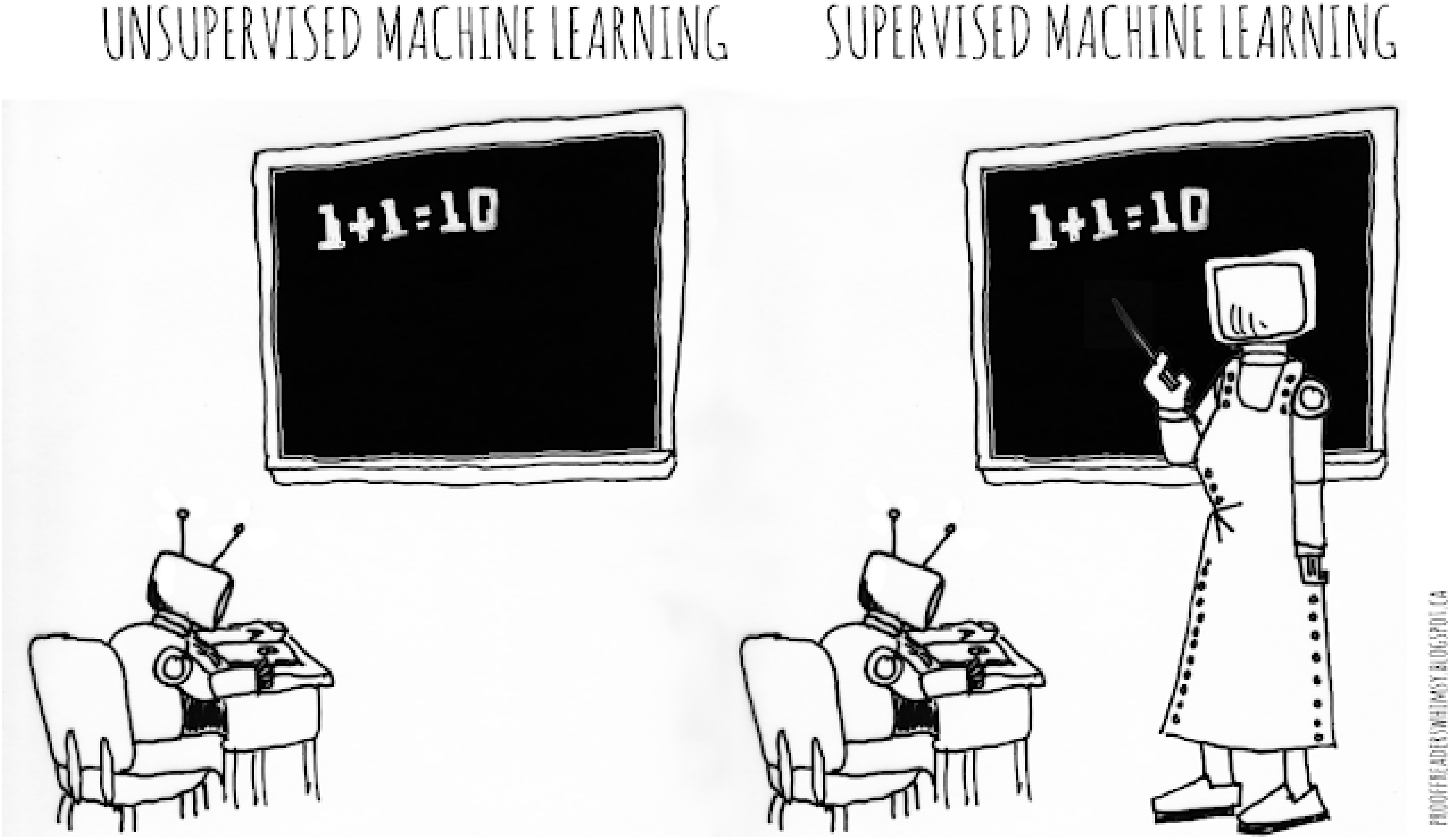
Jupyter notebook

- Logistic regression

Outline

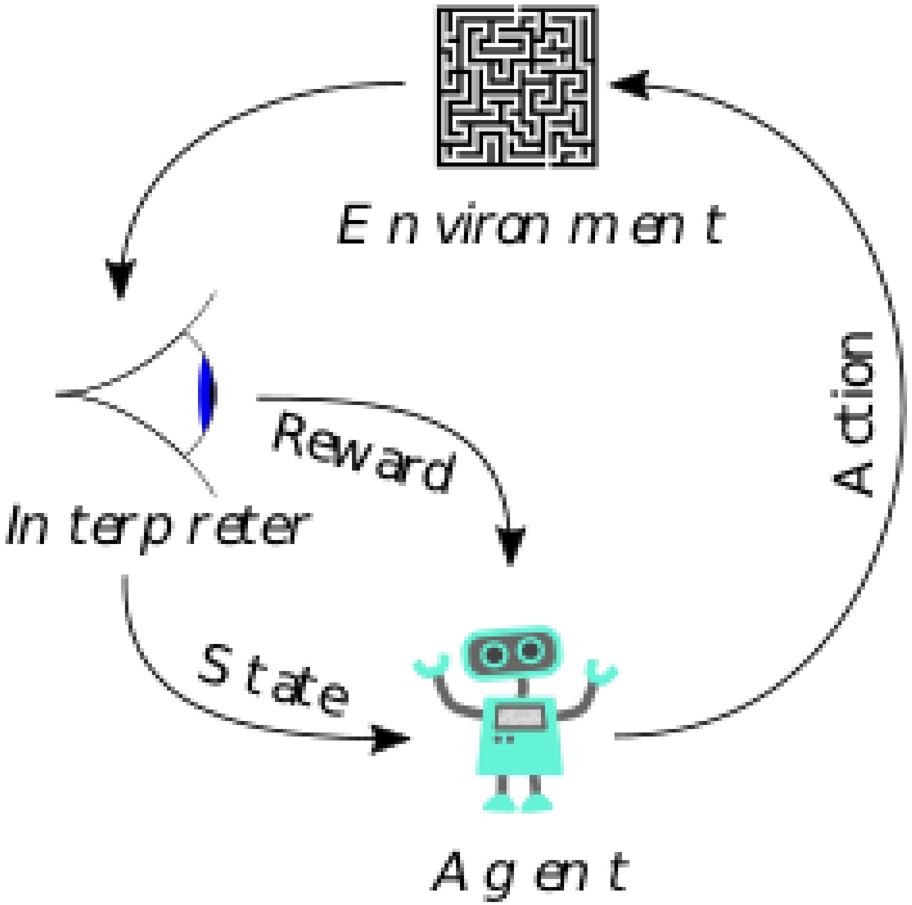
- ML 101: linear & logistic regression
- **Different learning paradigms and tasks**
- Neural Networks
- Clustering & Anomaly detection
- Advanced topics: CNNs and RL

Different learning paradigms



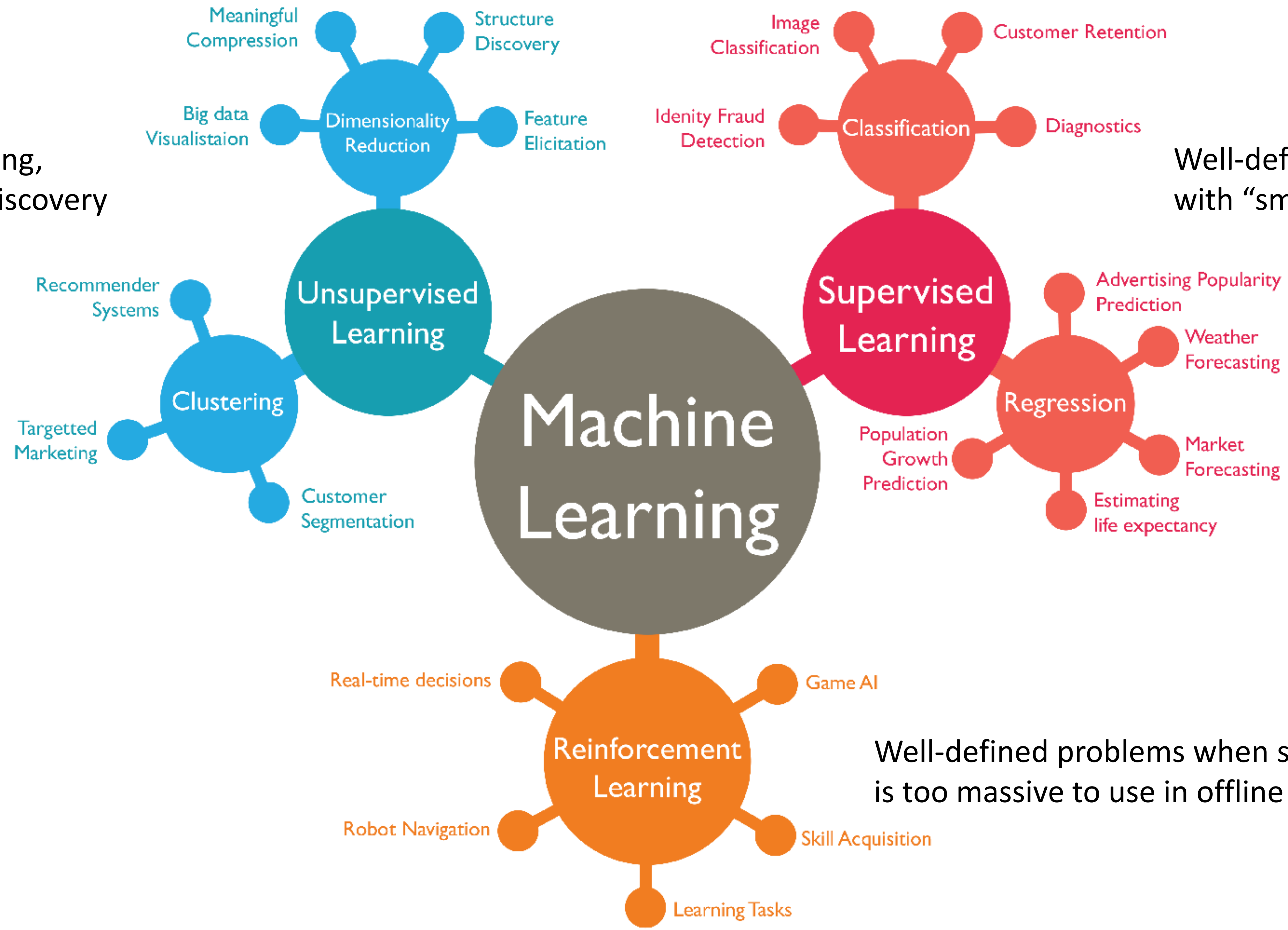
Different learning paradigms

- Reinforcement learning:



Data mining,
pattern discovery

Well-defined problems
with “small” search space



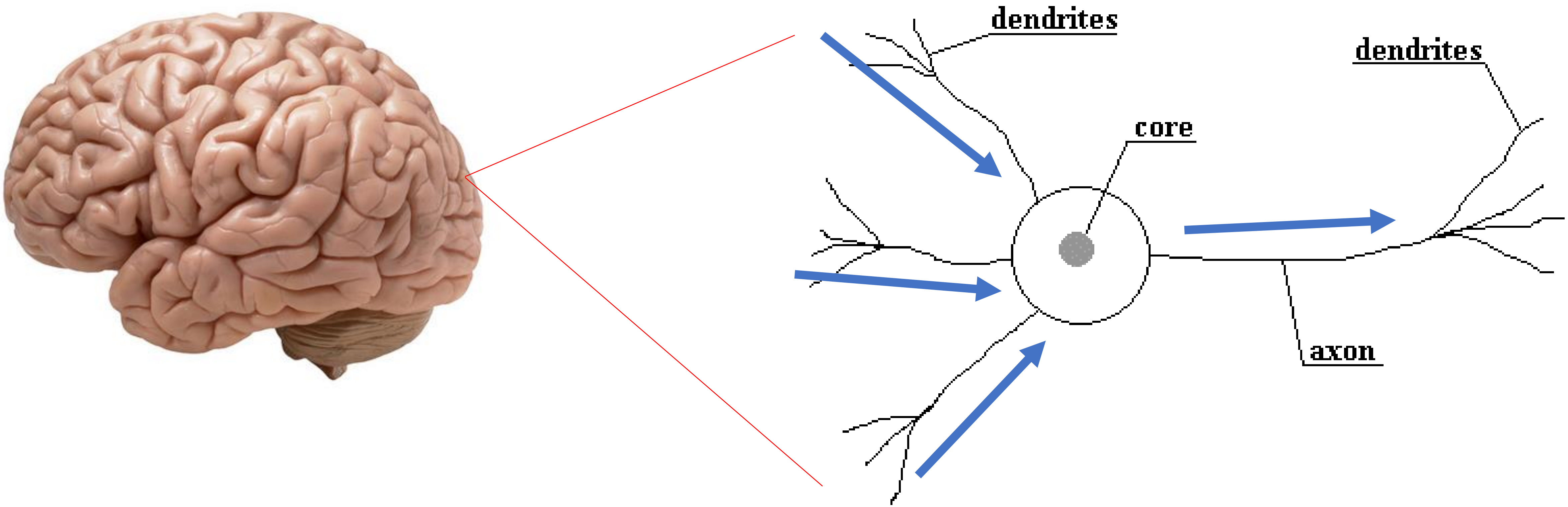
Well-defined problems when search space
is too massive to use in offline training

Outline

- ML 101: linear & logistic regression
- Different learning paradigms and tasks
- **Neural Networks**
- Clustering & Anomaly detection
- Advanced topics: CNNs and RL

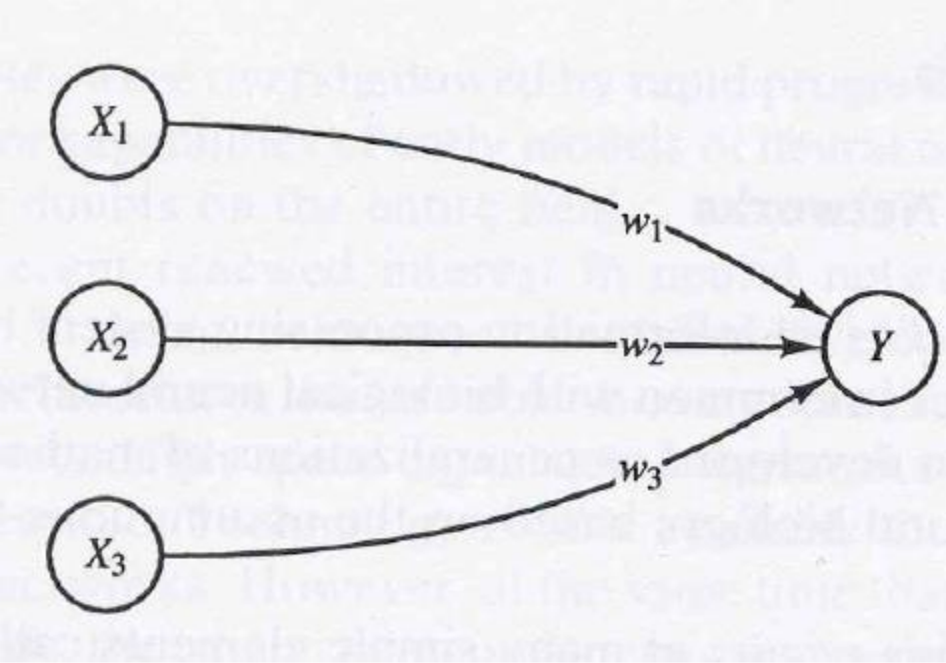
Biological Neural Networks

A neuron



A Simple Artificial Neuron

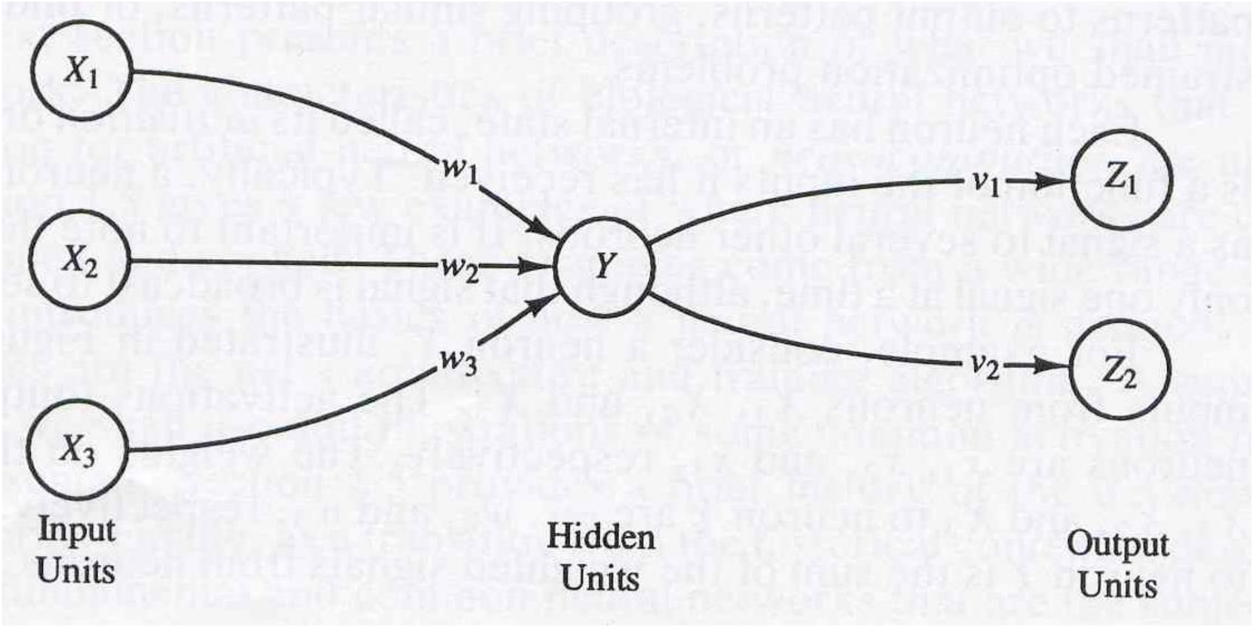
- A neuron *Y* receives inputs from neurons X_1, X_2 and X_3.
- The outputs from these neurons are x_1, x_2, x_3.
- The net input y_{in} to the neuron *Y* is the **sum of the weighted signals** from the neurons.



$$y_{in} = w_1 x_1 + w_2 x_2 + w_3 x_3$$

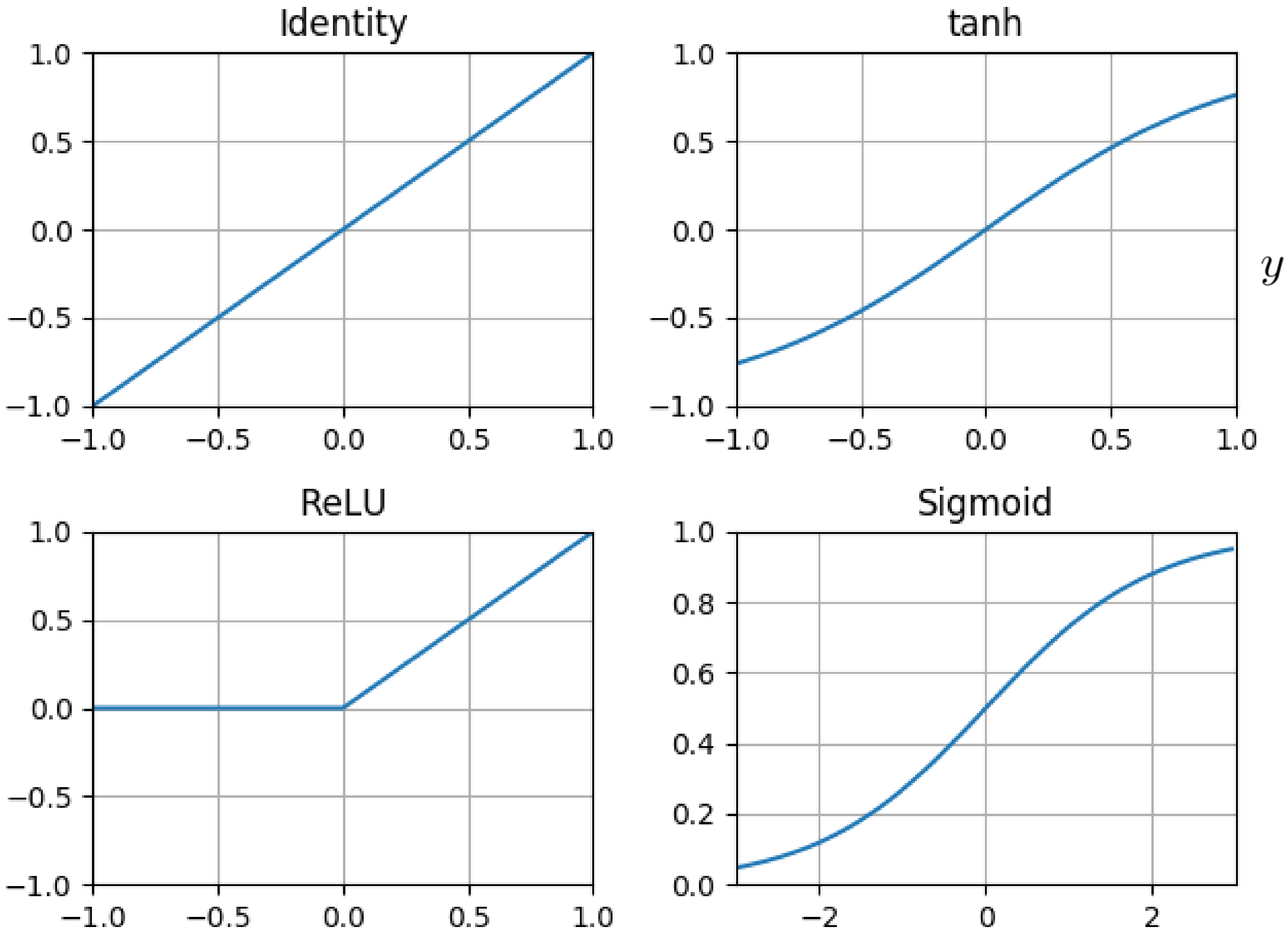
Typical ANN Architecture

- A further layer of neurons may be connected after neuron *Y*.
- In this case, the middle layer consisting of neuron *Y* is referred to as a 'hidden' layer.
- The output of $Y = f(y_{in})$, where f is called the *activation function*.



Different activation functions

$$y = x$$



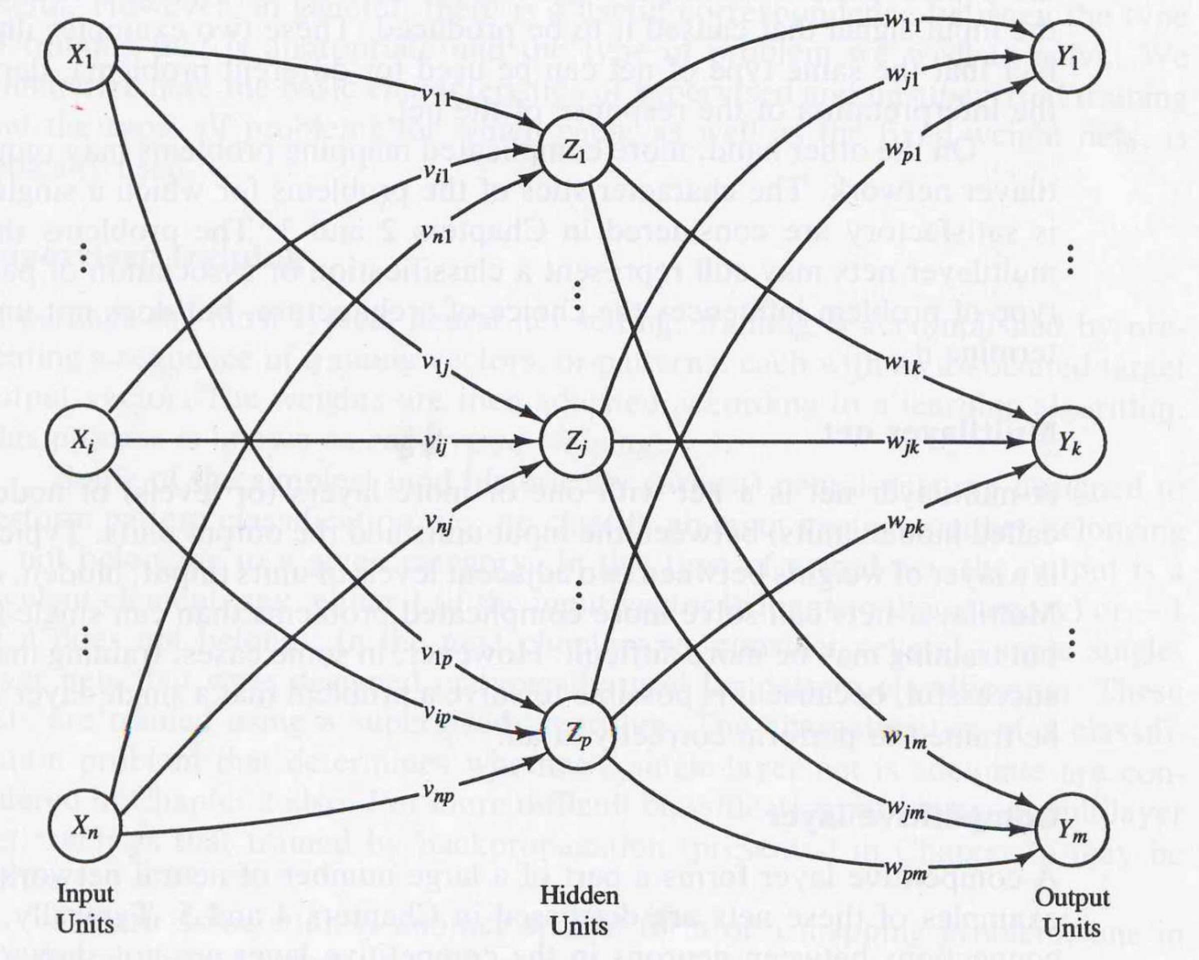
$$y = \tanh x = \frac{2}{1 + e^{-2x}} - 1$$

$$f(x) = \begin{cases} 0, & \text{if } x < 0. \\ 1, & \text{if } x \geq 0. \end{cases}$$

$$y = \frac{1}{1 + e^{-x}}$$

Typical ANN Architectures

- More complicated problems may require a **multilayer network**.



NN Training using Backpropagation

- Note the inclusion of a bias neuron in each layer (except the output)

- Analogous to the intercept when trying to fit e.g. y = ax + b

- Training involves three steps:
    - Feedforward
    - Backpropagation
    - Weights adjustment

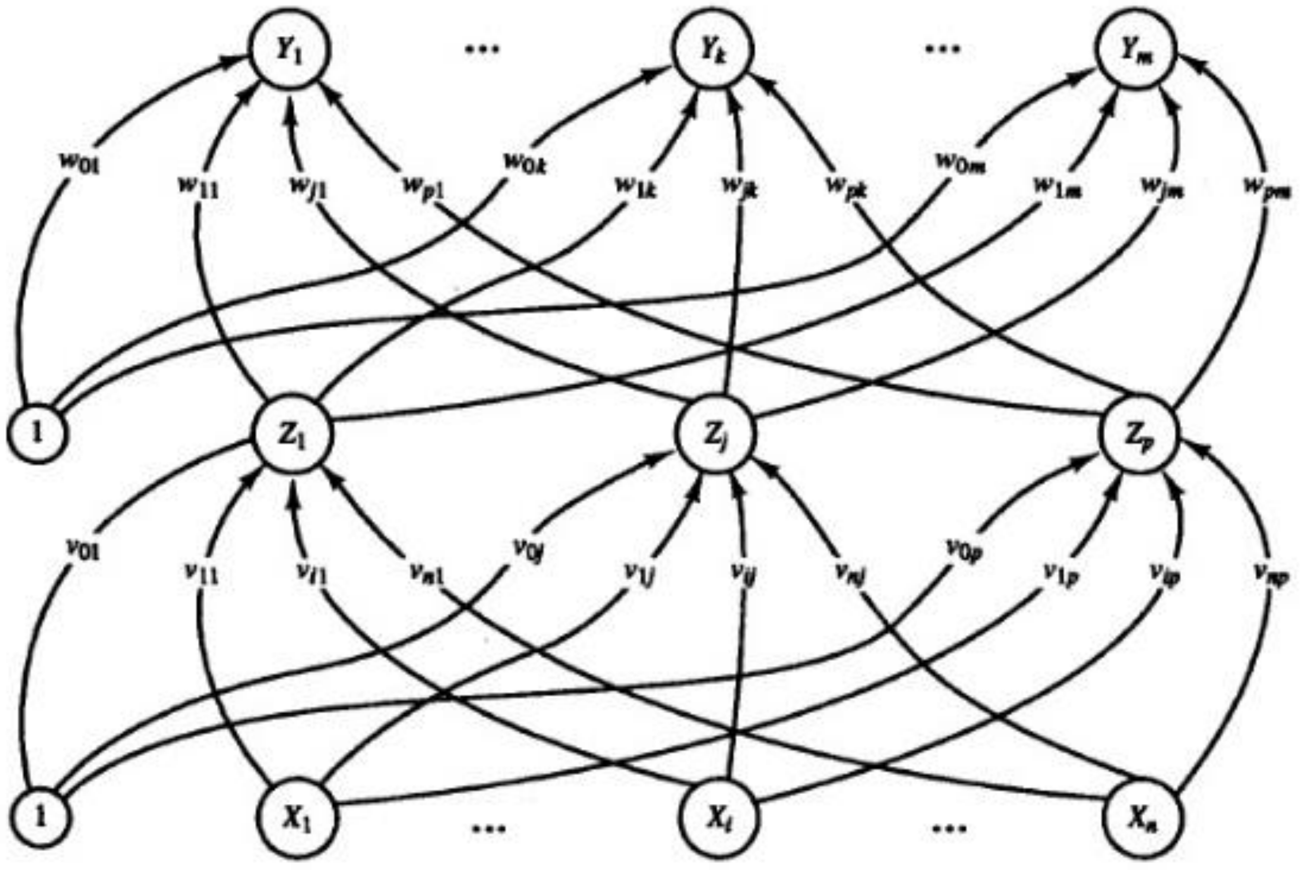
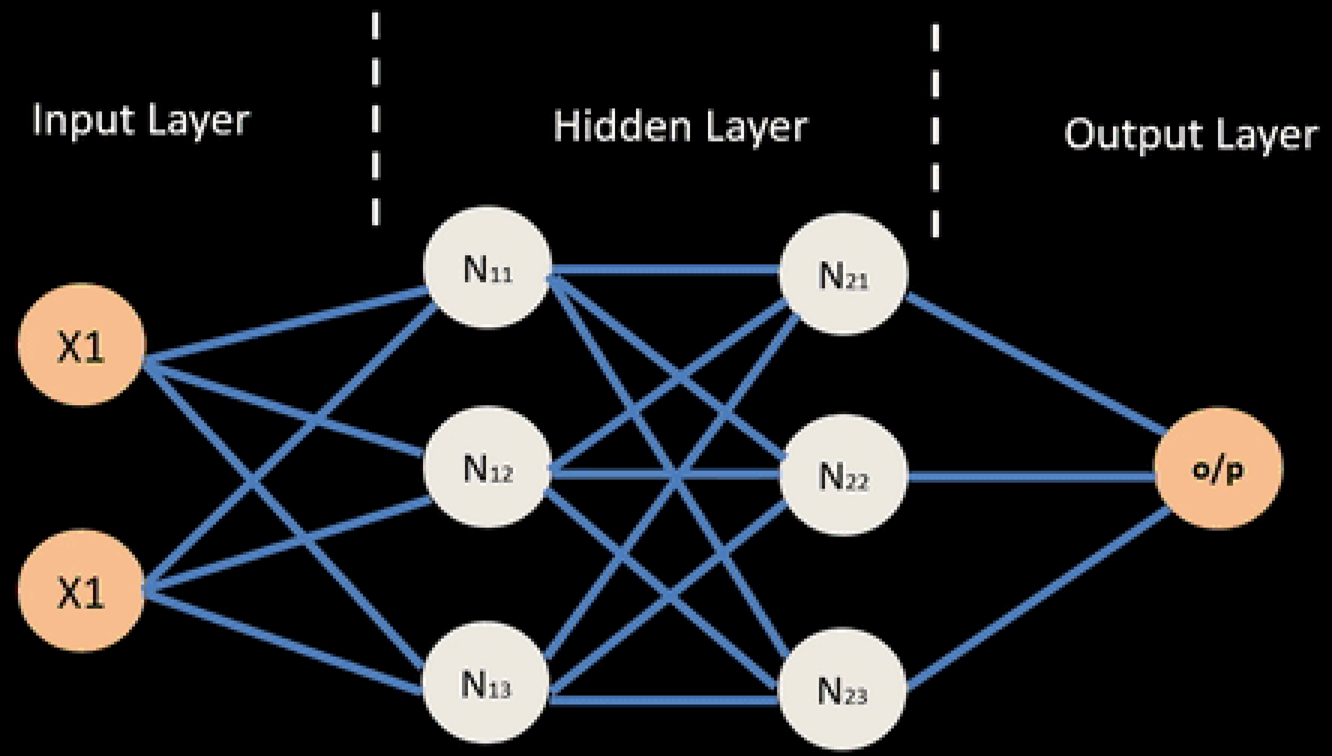


Figure 6.1 Backpropagation neural network with one hidden layer.

Neural Network – Backpropagation



© machinelearningknowledge.ai

Increasing the number of hidden layers

- A single hidden layer may not be sufficient for some problems.
- We can increase the number of hidden layers for as much is needed.

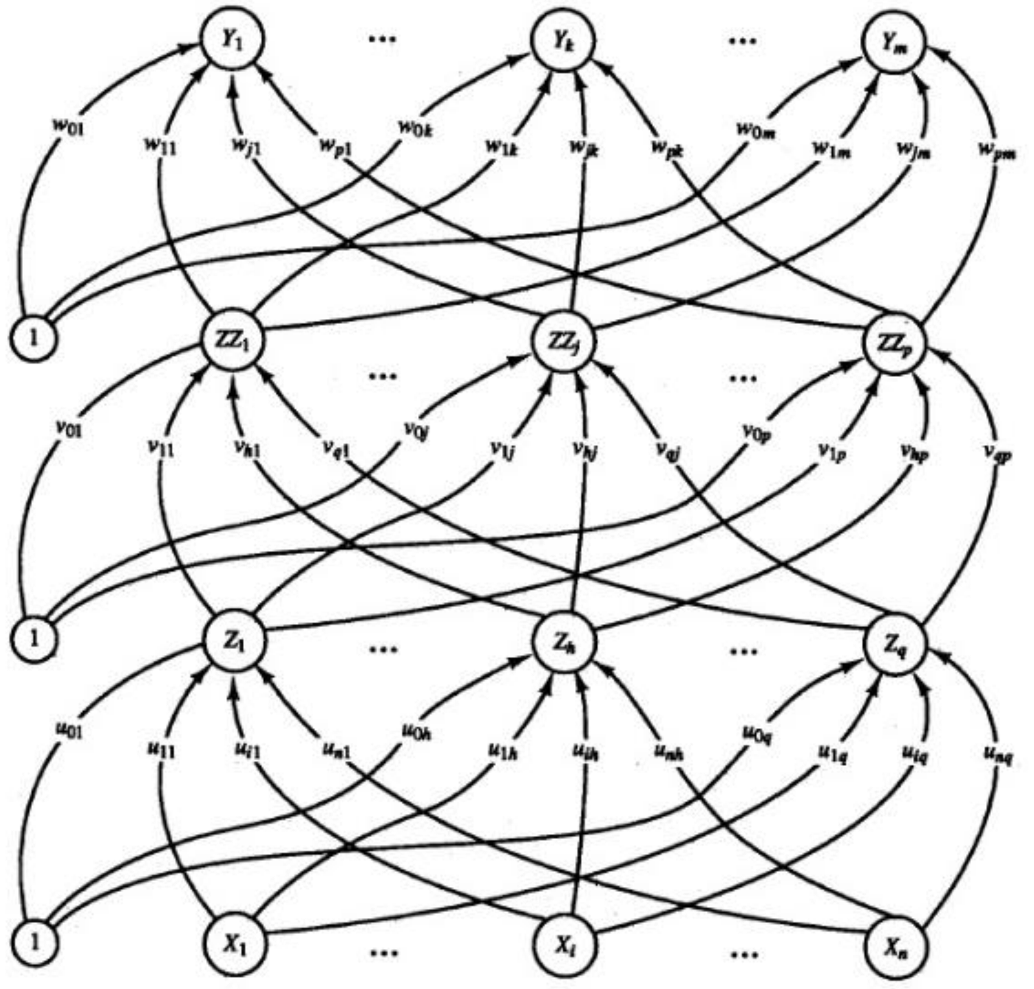


Figure 6.11 Backpropagation neural network with two hidden layers.

Hyperparameters (so far..)

- Number of neurons in each layer
- Number of layers
- Activation function
- Learning rate

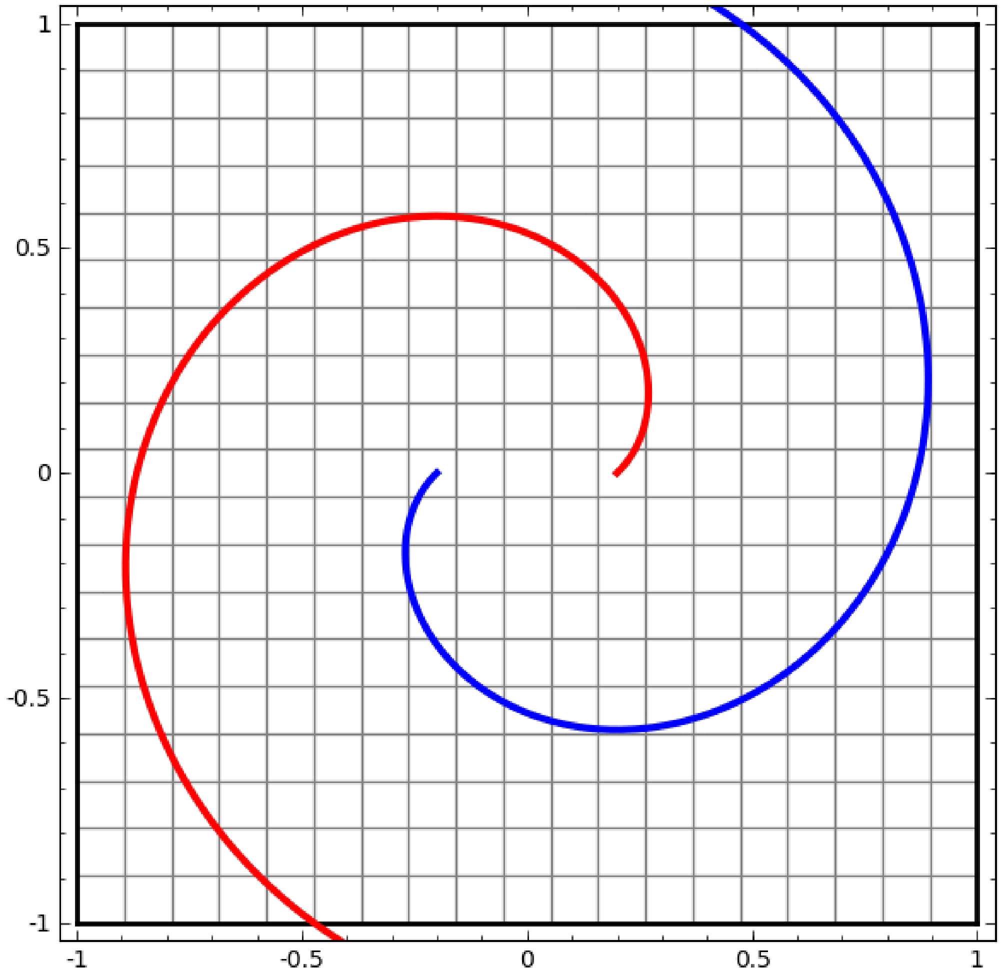
Spiral Classification using a NN

- Jupyter notebook

Spiral Classification

- First network:
  - Multiple linear layers -> can be rewritten as a single linear layer (i.e. a linear classifier)
  - We need a series of nonlinear layers to “warp the data”

Animation of warping by neural network



Spiral Classification

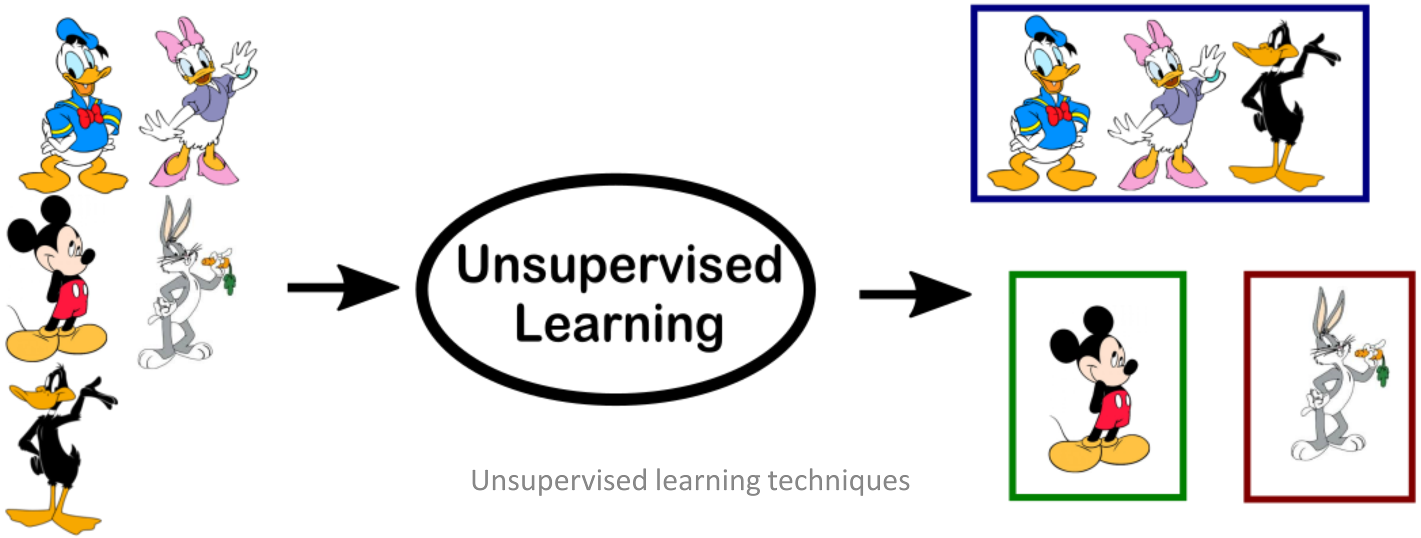
- Second network:
  - One hidden layer
  - Layers have logistic activation functions -> non-linear
  - Decision boundaries are now non-linear

Outline

- ML 101: linear & logistic regression
- Different learning paradigms and tasks
- Neural Networks
- **Clustering & Anomaly detection**
- Advanced topics: CNNs and RL

What is unsupervised learning?

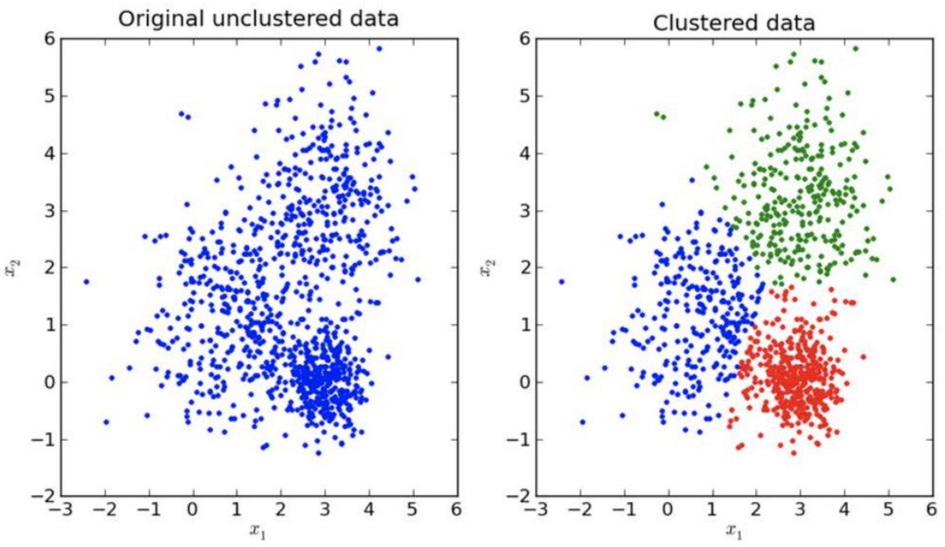
- As opposed to supervised learning which we have seen so far.
- Unsupervised: there are no labels / targets in the dataset.
- We are more interested in uncovering information in our data (*data mining*) rather than predicting new data.
- E.g. anomaly detection (fraud, equipment failure, medical problems..), market research, identifying patterns and groups of objects etc



Clustering

- **Clustering:** grouping a set of items together in such a way that items in one group (a cluster) are more similar to each other than to those in other groups.

- There are several types of clustering algorithms:
  - Hierarchical clustering (e.g. Linkage clustering)
  - Centroid-based clustering (e.g. K-Means)
  - Distribution-based clustering (e.g. Expectation-Maximization)
  - Density-based clustering (e.g. DBSCAN)



K-Means clustering algorithm

- Suppose we have a dataset x_1, x_2, x_3, ..., x_N} consisting of *N* observations of *D* dimensional vectors **x** (i.e. *D* features).

- The goal is to partition the dataset into *K* clusters.
  - Therefore, the number of clusters in our dataset needs to be known a priori.

- A cluster is a group of data points whose distances between one another in *D*-dimensional space are small compared to points outside the cluster.

- This can be formalized by introducing a *D*-dimensional mean vector μ_k, where $k = 1,2,3,...K$.
  - This represents the center of the cluster.

K-Means clustering algorithm

- The K-means clustering algorithm assigns a vector $x_{i,j}$ to the cluster which minimizes the distortion measure: $J_k = ||x_{i,j} - \mu_k||^2$

- The mean vector is then updated by computing the mean intensity value of the considered cluster such that:

$$\mu_k = \frac{\sum_i \sum_j r_{i,j,k} x_{i,j}}{\sum_i \sum_j r_{i,j}}$$

where:

$$r_{i,j,k} = \begin{cases} 1, & \text{if } k = arg_k min(||x_{i,j} - \mu_k||^2) \\ 0, & \text{otherwise.} \end{cases}$$

Clustering

- Jupyter notebook

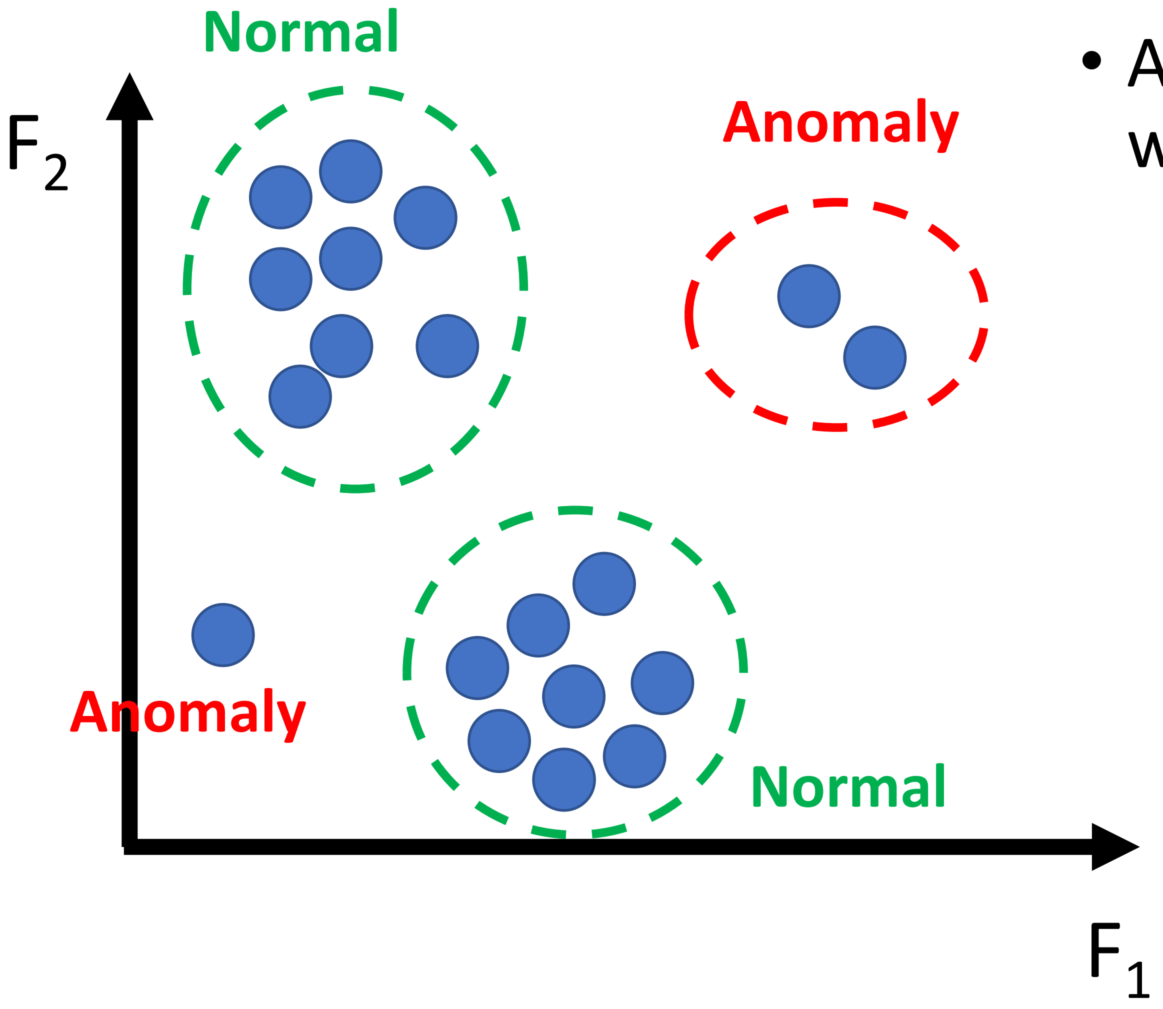
Anomaly Detection

- The process of determining which points in a dataset are *different* than *most of* the others.

- Types of anomalies:

  - Point anomalies
  - Contextual anomalies
  - Collective anomalies

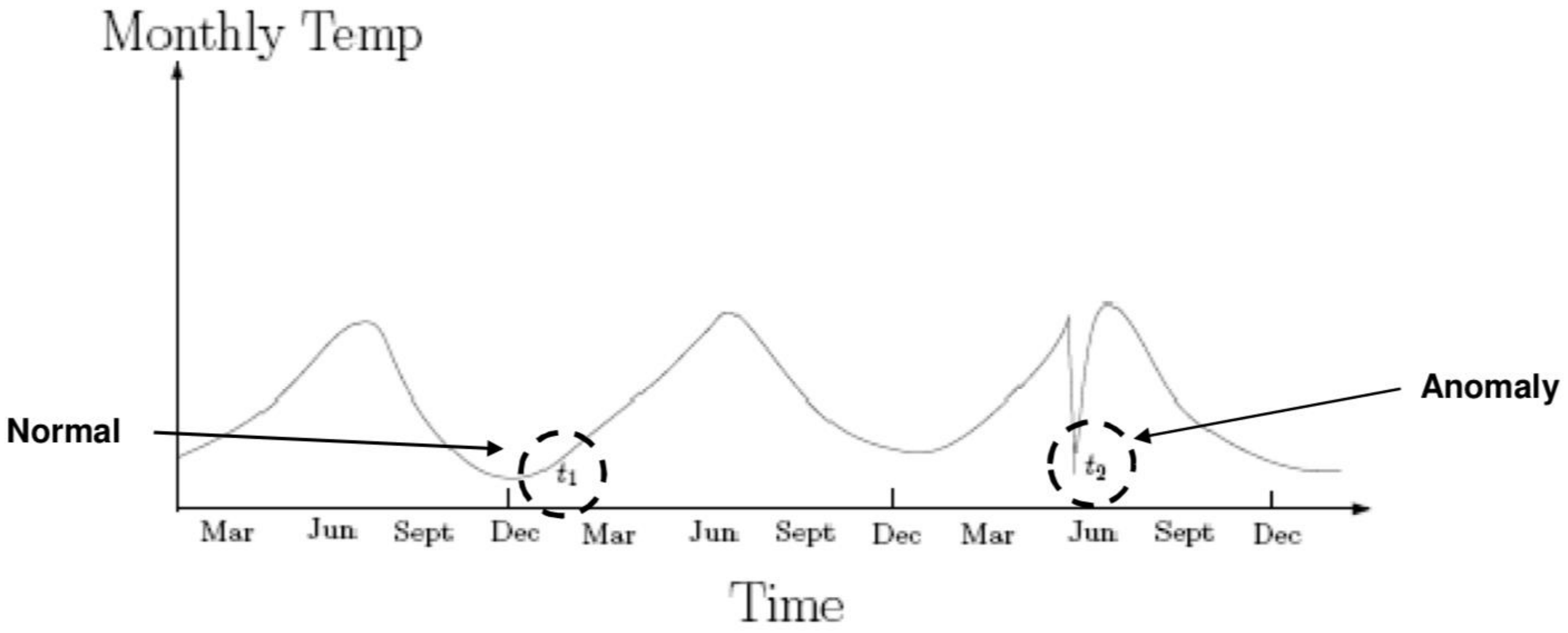
Point anomalies



- An individual data **point** is anomalous with respect to the surrounding data

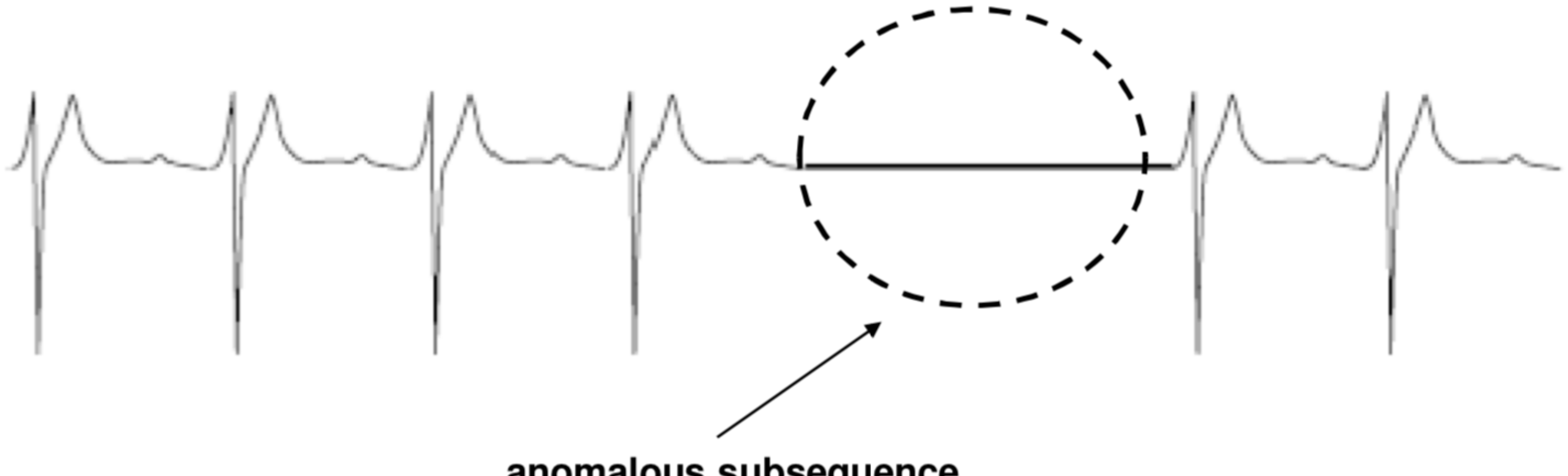
Contextual anomalies

- An individual data instance is anomalous within a **context**
- Also referred to as a conditional anomaly



Collective anomalies

- A **collection** of related data instances is anomalous
- Requires a relationship among data instances
  - Sequential data
  - Spatial data
  - Graph data
- The individual instances within a collective anomaly are not anomalous by themselves



Anomaly Detection

- In anomaly detection, we want to identify outliers which do not resemble the bulk of the dataset.

- Note that we may use supervised learning techniques for anomaly detection (e.g. a dataset which was previously labelled as ”normal” or “abnormal”)
  - This would be a 2-class classification problem
  - ..but introduces issues due to the expected class imbalance

- There are a variety of techniques (for point based):
  - Distance based methods (k nearest neighbours)
  - Density based methods (local outlier factor)
  - One-class SVMs
  - Clustering

- For time-series data:
  - LSTM autoencoders
  - Transformer models

Anomaly Detection – kNN distance

- Compute an outlier score as distance to k^{th} nearest neighbor
- Score is sensitive to choice of k

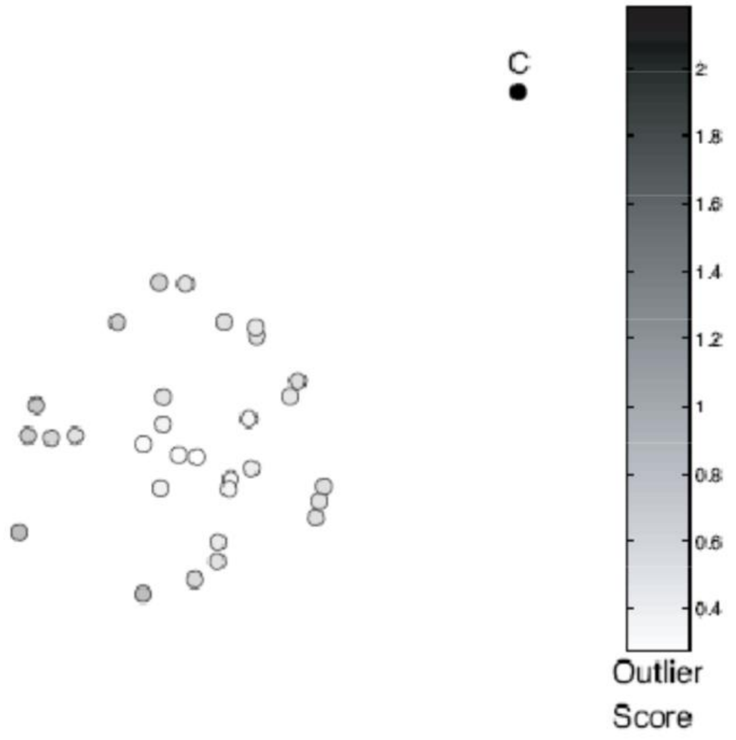


Figure 10.4. Outlier score based on the distance to fifth nearest neighbor.

Anomaly Detection – kNN distance

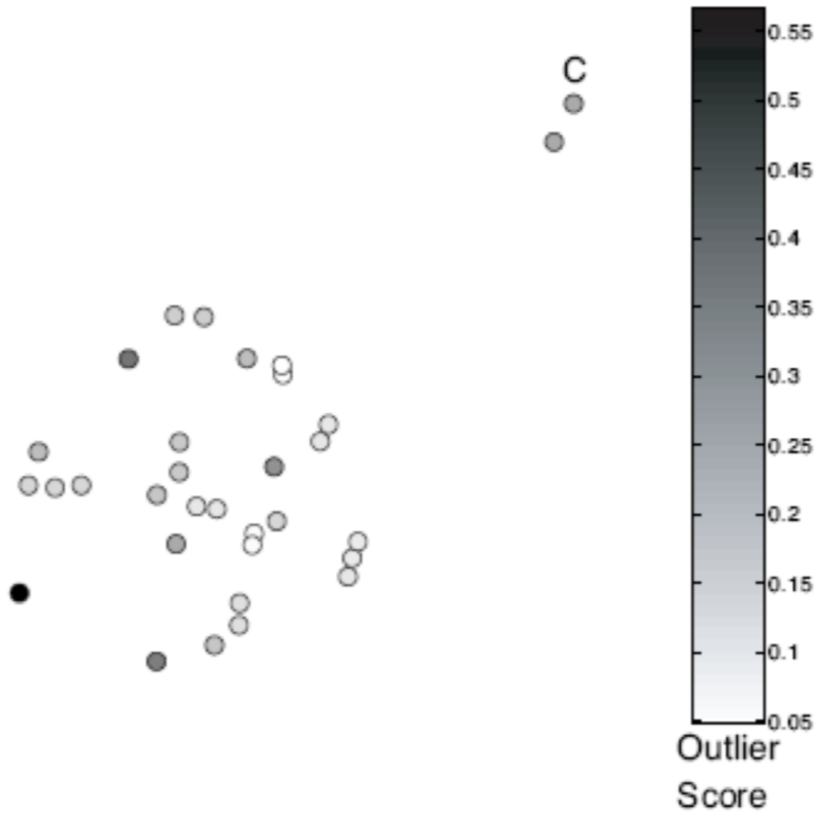


Figure 10.5. Outlier score based on the distance to the first nearest neighbor. Nearby outliers have low outlier scores.

Anomaly Detection – kNN distance

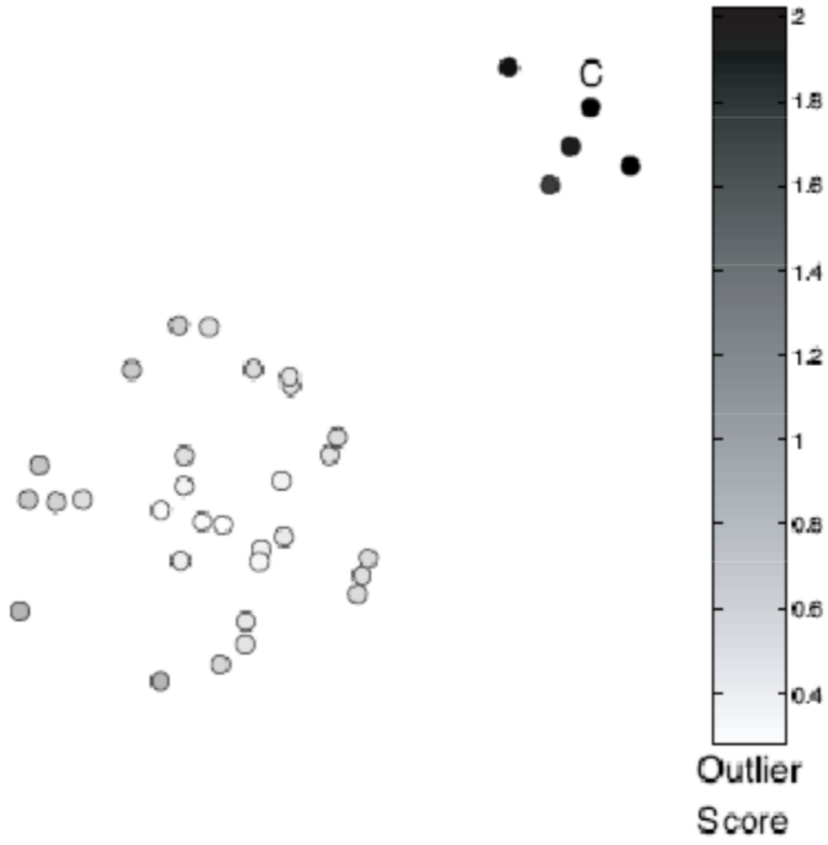


Figure 10.6. Outlier score based on distance to the fifth nearest neighbor. A small cluster becomes an outlier.

Anomaly Detection – kNN distance

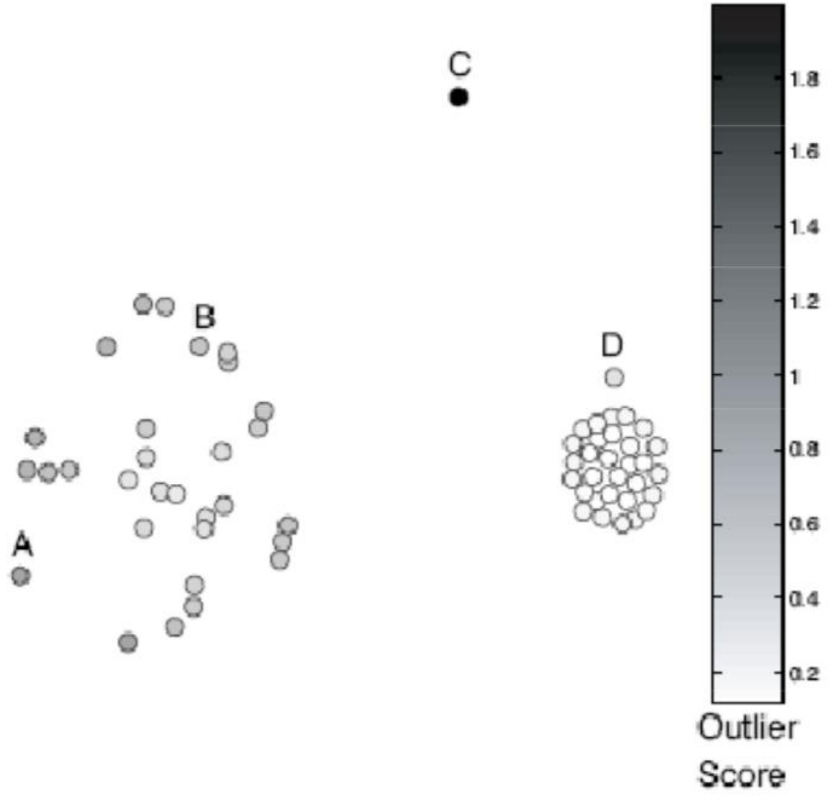


Figure 10.7. Outlier score based on the distance to the fifth nearest neighbor. Clusters of differing density.

Local Outlier Factor

- One of the most popular anomaly detection algorithms (proposed > 20 years ago).
- **Local:** is able to find local anomalies.
- Basic idea:

1. Find the k-nearest neighbours
2. For each instance, compute the *local reachability density* (LRD):

$$\mathrm{lrd}(A) := 1 / \left(\frac{\sum_{B \in N_k(A)} \text{reachability-distance}_k(A, B)}{|N_k(A)|} \right)$$

where: - $N_k(A)$ is the set of k nearest neighbours of A
- reachability-distance$_k$(A, B) is the maximum between (a) the distance of A and B, or (b) the k-distance of B (i.e. the distance of B to its own k^{th} nearest neighbour.
- $|N_k(A)|$ is the cardinality of the set.

Local Outlier Factor

3. For each instance, compute the ratio of local densities to obtain the local outlier factor (LOF):

$$\mathrm{LOF}_k(A) := \frac{\sum_{B \in N_k(A)} \frac{\mathrm{lrd}(B)}{\mathrm{lrd}(A)}}{|N_k(A)|} = \frac{\sum_{B \in N_k(A)} \mathrm{lrd}(B)}{|N_k(A)|} / \mathrm{lrd}(A)$$

- This is therefore the *average local reachability density of the neighbours* divided by the object's own local reachability density.

- LOF ~ 1 indicates that an object is comparable to its neighbours (not outlier)

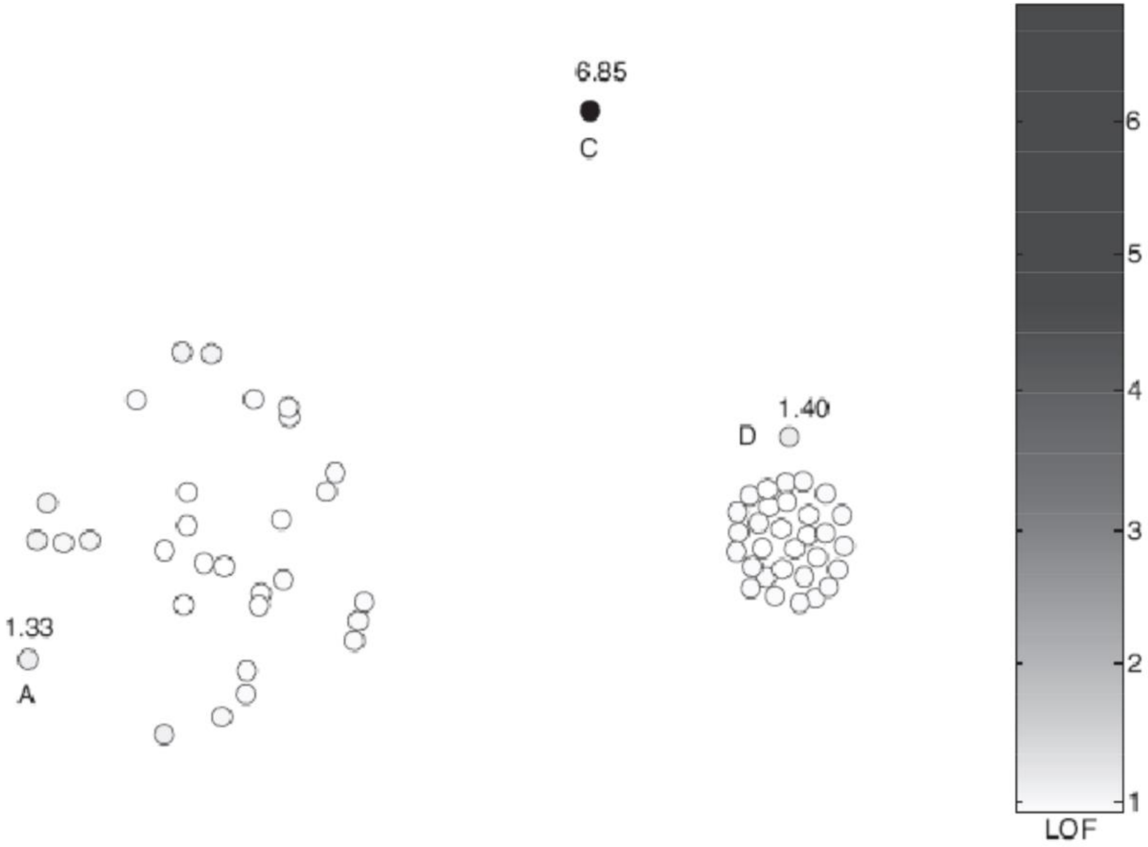
Local Outlier Factor

- A rule of thumb: the number of neighbours considered is typically chosen:
    1. greater than the minimum number of objects a cluster has to contain, so that other objects can be local outliers relative to this cluster;
    2. smaller than the maximum number of close by objects that can potentially be local outliers.

- This info is generally not available a priori, but taking k = 20 seems to work well in general.

- The larger the LOF score, the more likely it is that a data point is an outlier.

Local Outlier Factor



relative density (LOF) outlier scores

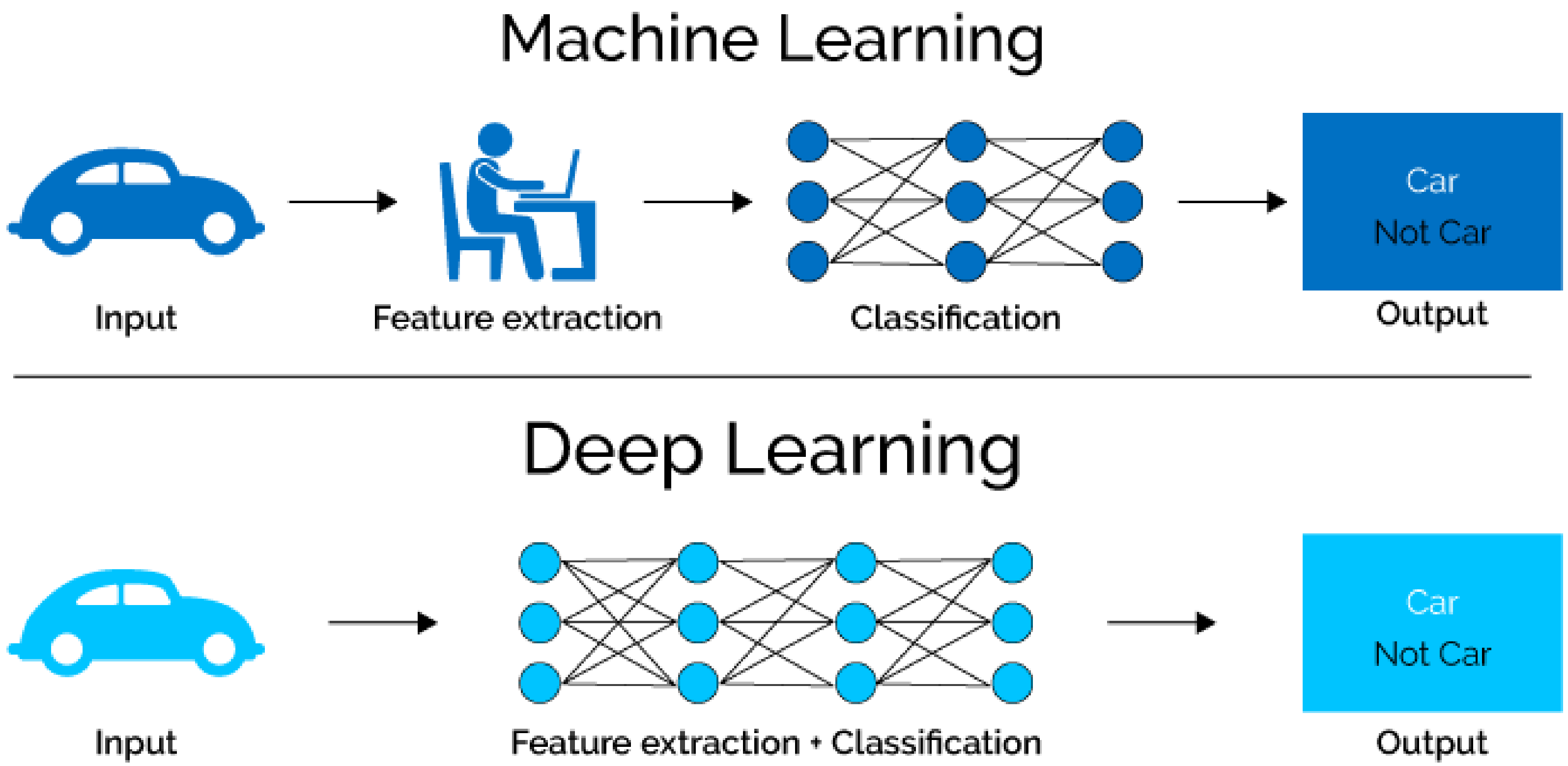
Anomaly Detection

- Jupyter notebook

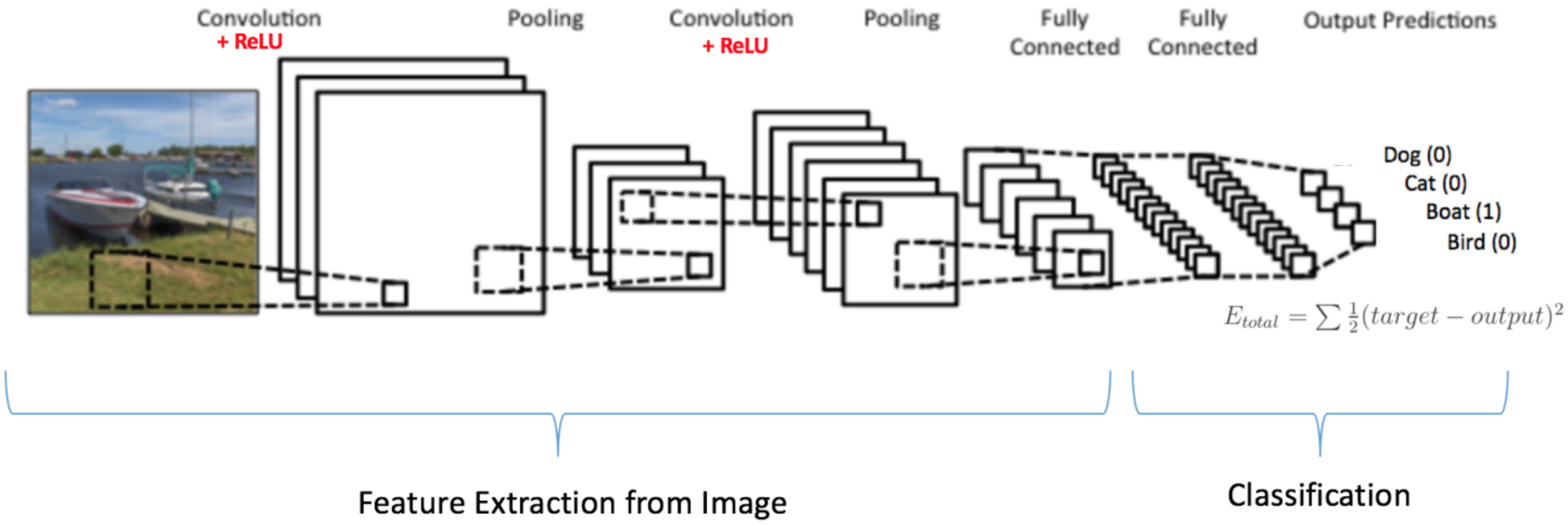
Outline

- ML 101: linear & logistic regression
- Different learning paradigms and tasks
- Neural Networks
- Clustering & Anomaly detection
- **Advanced topics: CNNs and RL**

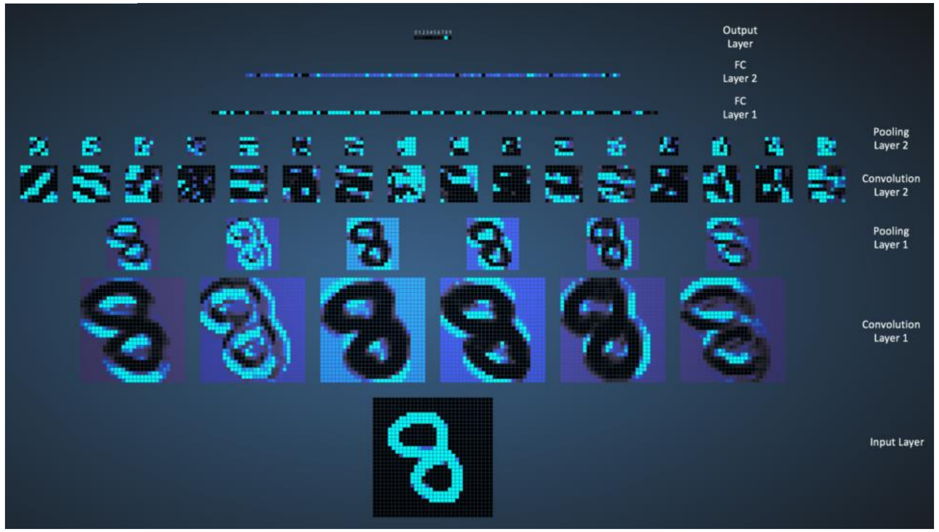
CNNs: ML vs Deep Learning



Convolutional Neural Network



- Visualizing a CNN:



Edge Detection using the convolution operator

Suppose we have a vertical edge in our image

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

*

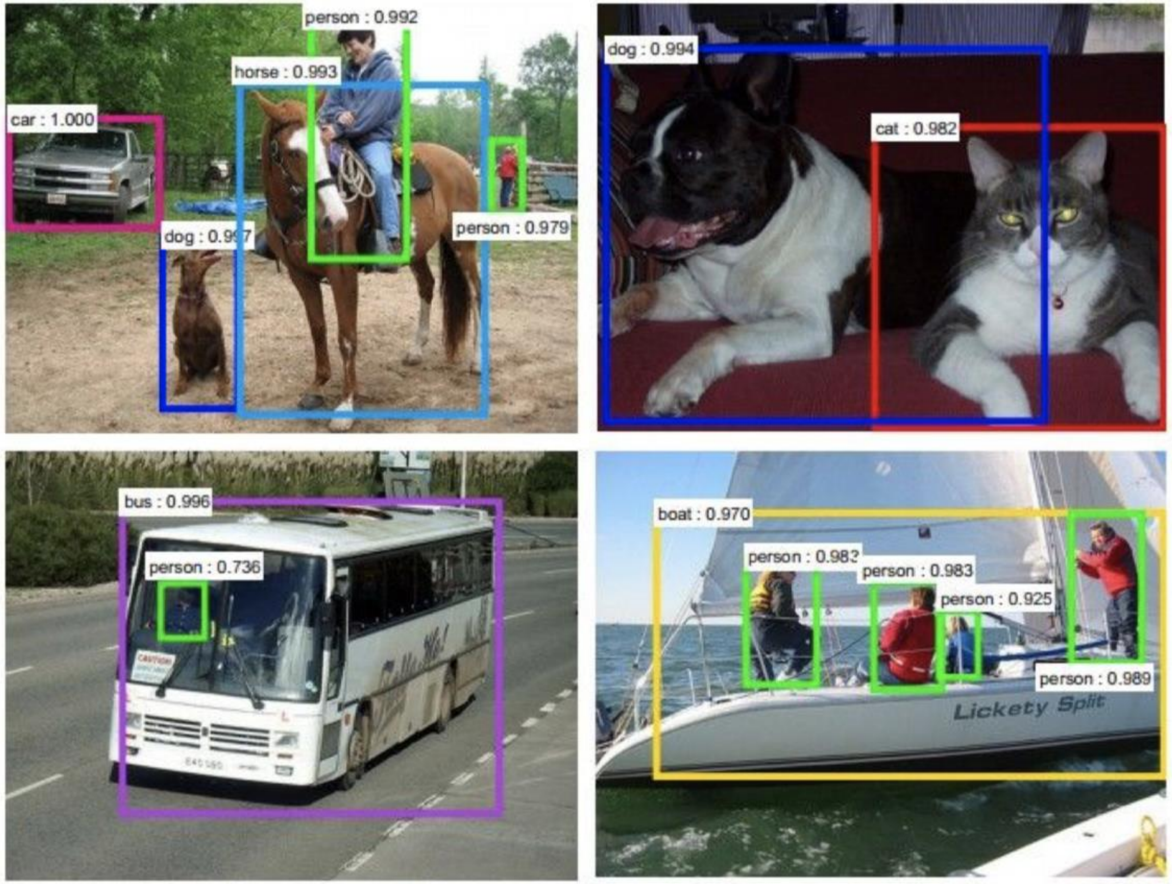
filter

1	0	-1
1	0	-1
1	0	-1

=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

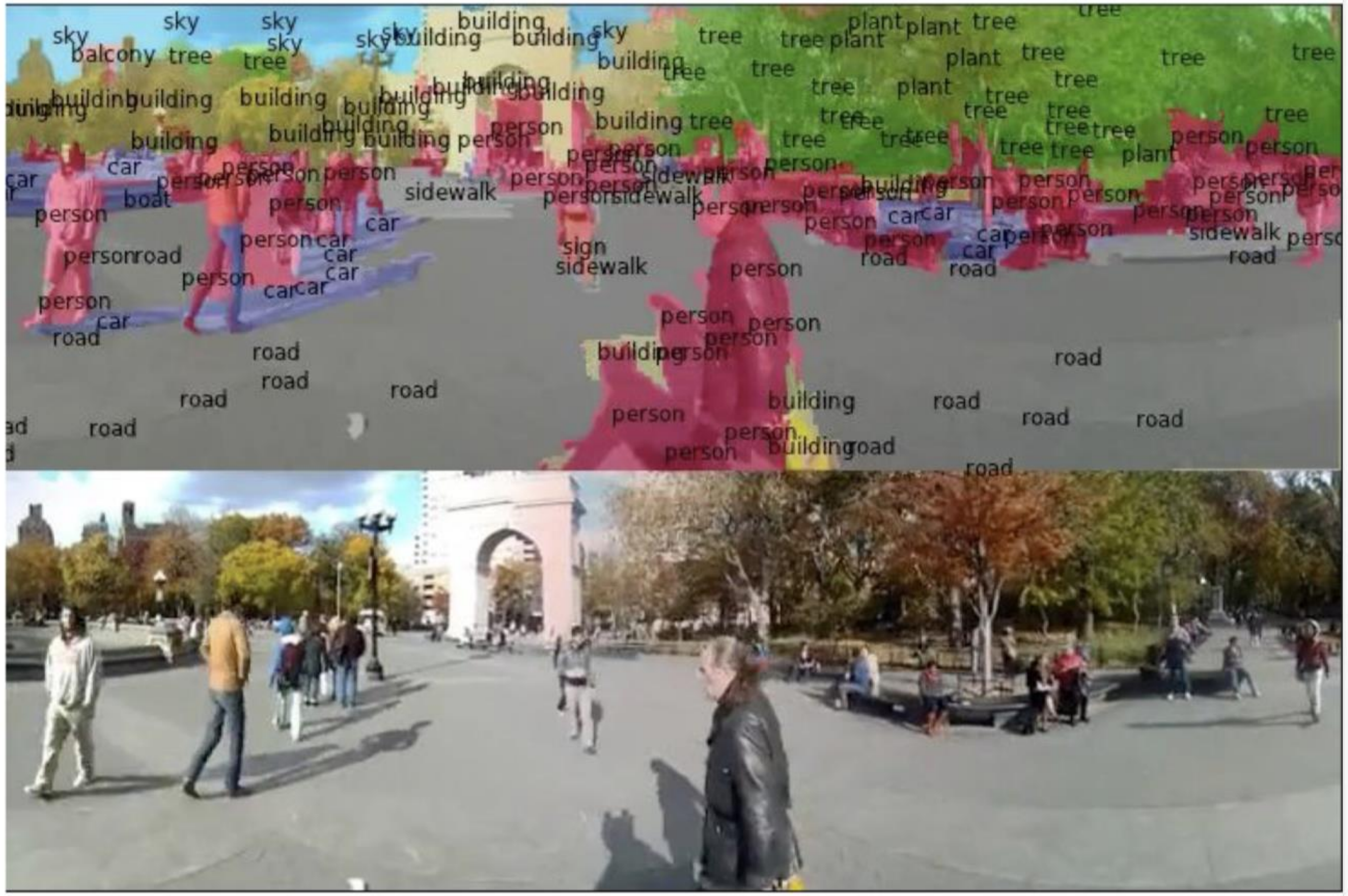
Detection



Figures copyright Shaoqing Ren, Kaiming He, Ross Girschick, Jian Sun, 2015. Reproduced with permission.

[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation

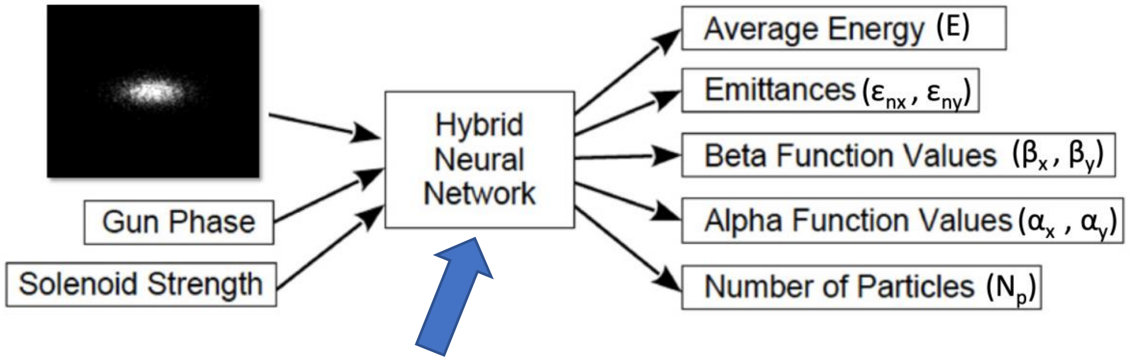


Figures copyright Clement Farabet, 2012. Reproduced with permission.

[Farabet et al., 2012]

Image-based diagnostics using CNNs

- **Objective:** predict beam parameters given input beam distribution, gun phase and solenoid strength at FAST facility.
- PARMELA simulation data of first 8 m of FAST low energy beamline used.



CNN: used to extract features from virtual cathode image
NN: combines these features together with gun phase and solenoid strength

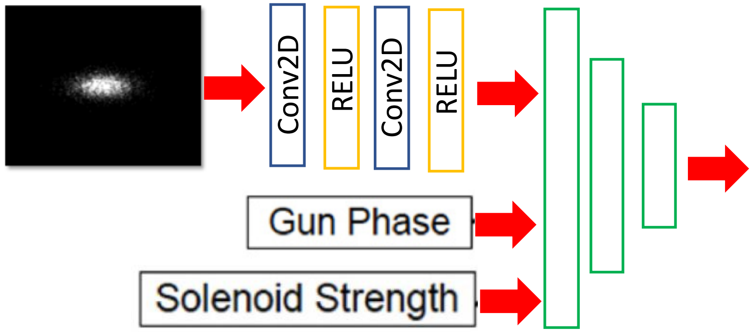


Table 1: Max and Min Values for Predicted Parameters

Param.	Max Gun	Min Gun	Max CC2	Min CC2
N_p	5001	1015	5001	1004
ε_{nx} [m-rad]	2.5e-4	1.6e-6	4.0e-4	9.1e-7
ε_{ny} [m-rad]	2.4e-4	1.6e-6	4.0e-4	8.5e-7
α_x [rad]	14.1	-775.1	0.8	-149.8
α_y [rad]	14.5	-797.0	0.7	-154.5
β_x [m/rad]	950.4	7.9e-2	820.2	0.7
β_y [m/rad]	896.8	8.4e-2	845.7	0.81
E [MeV]	4.6	3.2	47.2	42.8

Table 3: Model Performance at CC2 Exit

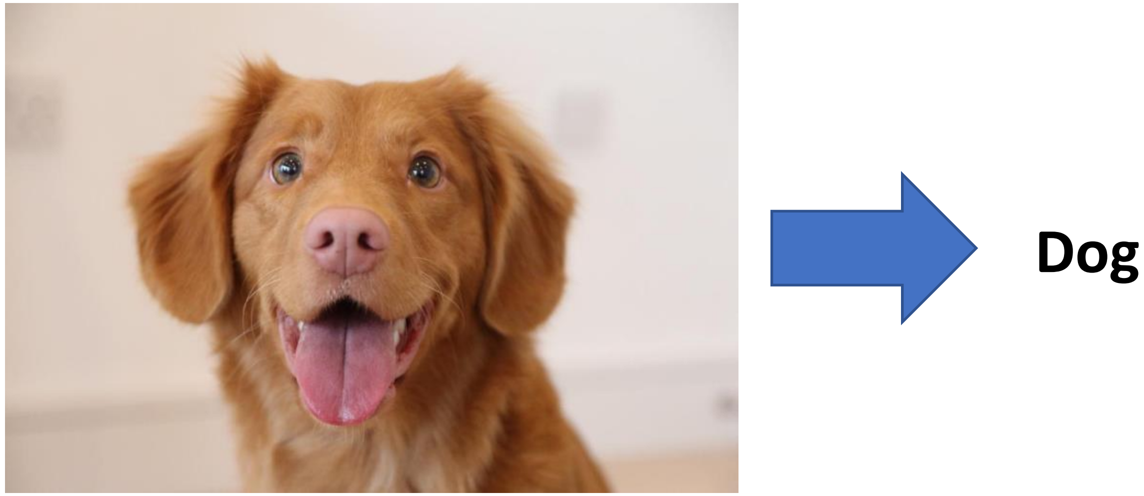
Param.	Train. MAE	Train. STD	Val. MAE	Val. STD
N_p	103.7	141.2	123.3	176.8
ε_{nx}	1.0e-5	1.2e-5	1.2e-5	1.6e-5
ε_{ny}	1.0e-5	1.3e-5	1.2e-5	1.5e-5
α_x	3.4	6.6	3.1	5.9
α_y	3.4	6.6	3.1	5.9
β_x	16.3	33.5	14.7	27.8
β_y	16.4	33.6	14.8	27.5
E	4.0e-2	3.9e-2	4.6e-2	6.2e-2

Training set size: 894
Validation set size: 600

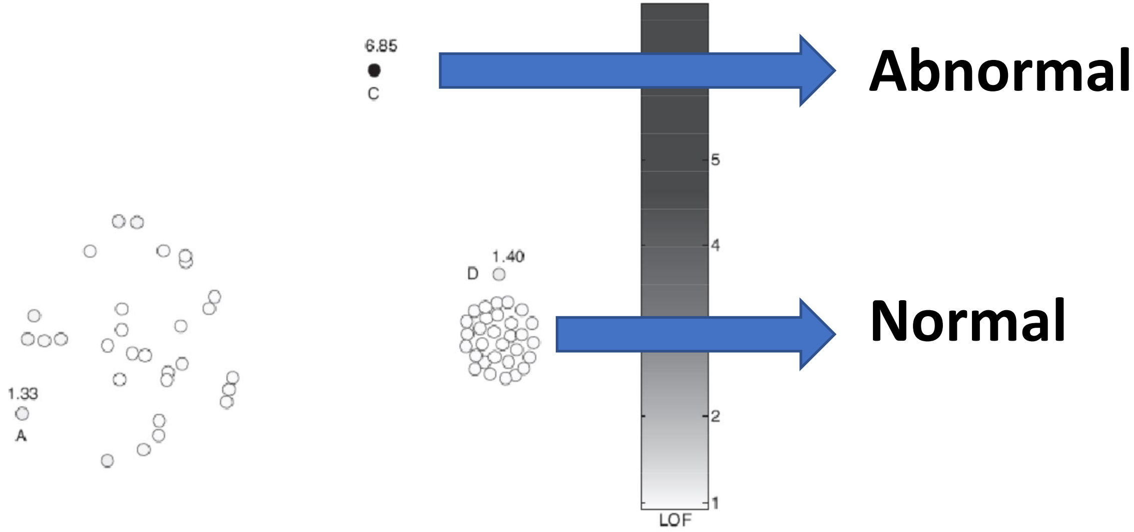
A. Edelen et al., Proc. NAPAC2016.

What is Reinforcement Learning?

- So far: **Supervised Learning**
  - **Data:** (X, y)
  - **Goal:** Learn a function to map X -> y
  - **Examples:** classification, regression, object detection etc

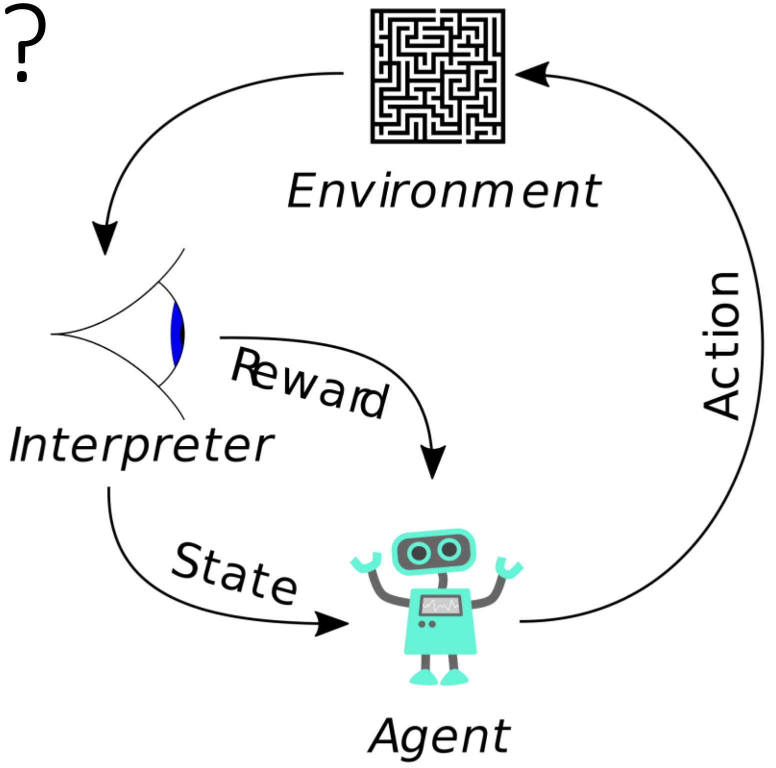


- So far: **Unsupervised Learning**
  - **Data**: X (no y)
  - **Goal:** Learn some underlying hidden structure in the data
  - **Examples:** clustering, dimensionality reduction, anomaly detection



relative density (LOF) outlier scores

What is Reinforcement Learning?



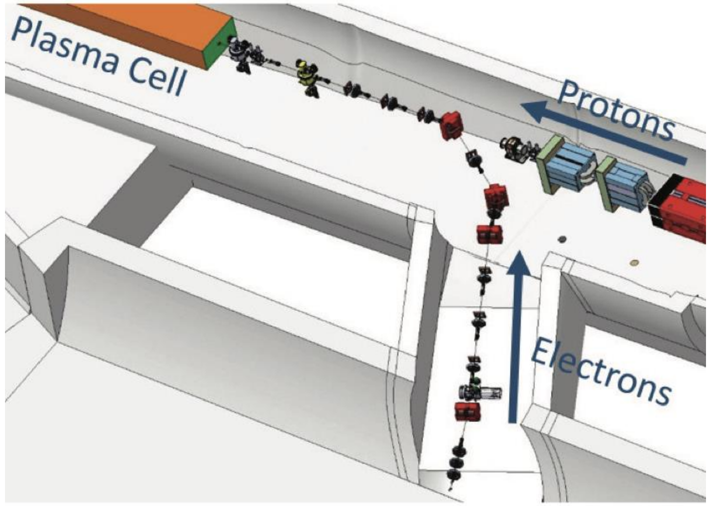
- In Reinforcement Learning, an **agent** interacts with an **environment** to learn how to perform a particular task **well**.

- How is it different to the other learning paradigms?
  - There is no supervisor, only a **reward.**
  - The agent's actions **affect the subsequent data it receives**
  - **Feedback is delayed**, and may be received after several actions

Examples of Reinforcement Learning

Fly a helicopter

Ensure a corrected orbit



Manage an investment portfolio

Play Atari games better than humans

Rewards

- The agent receives feedback from the environment through reward
- A reward R_t is a scalar feedback signal
- It is an indication of how well the agent is doing at step t
- The agent's job is to **maximise cumulative reward**
- Examples:
  - Winning a game
  - Achieving design luminosity in a collider
  - Maintaining an inverted pendulum at the top

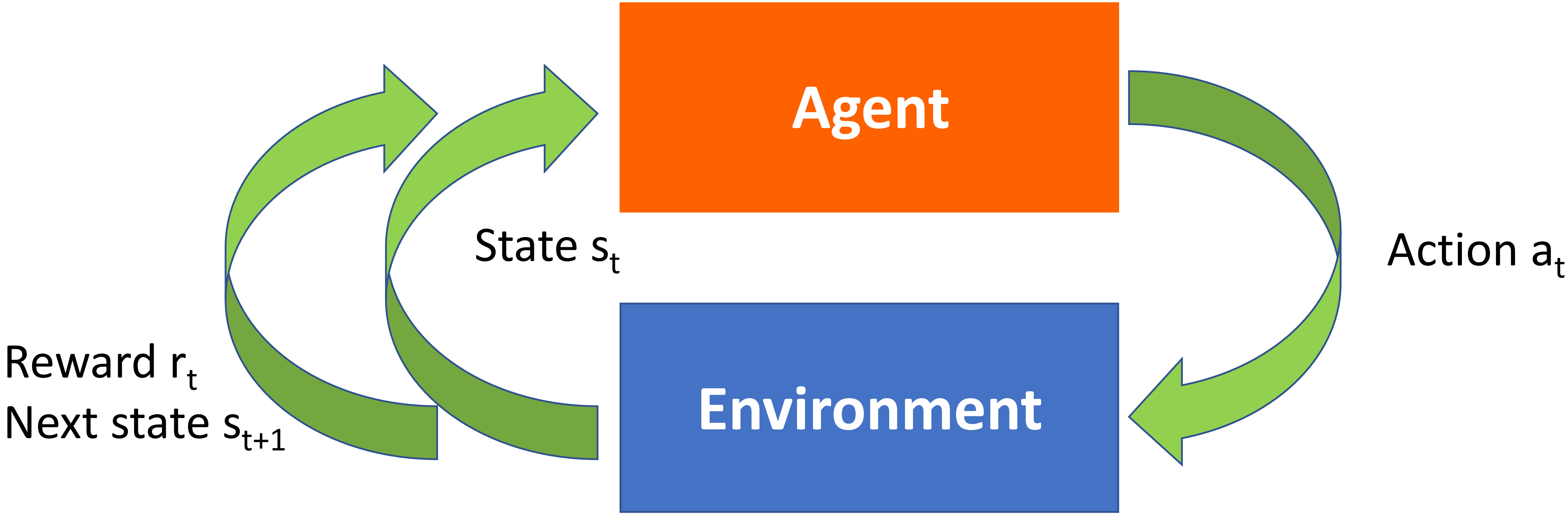
Sequential decision making

- **Goal:** select actions to maximise total future reward
- Actions may have long term consequences
- Reward may be delayed
- It may be better to sacrifice immediate reward to gain more long-term reward
- Examples:
  - A financial investment (may take months to mature)
  - Blocking opponent moves (might help winning probability many moves from now)

States

- **State:** what the agent is observing about the environment
- Examples:
  - Pixels in an image (of a game, of a driverless car, etc)
  - Data from beam instrumentation in an accelerator
  - The position of all pieces in a game of chess

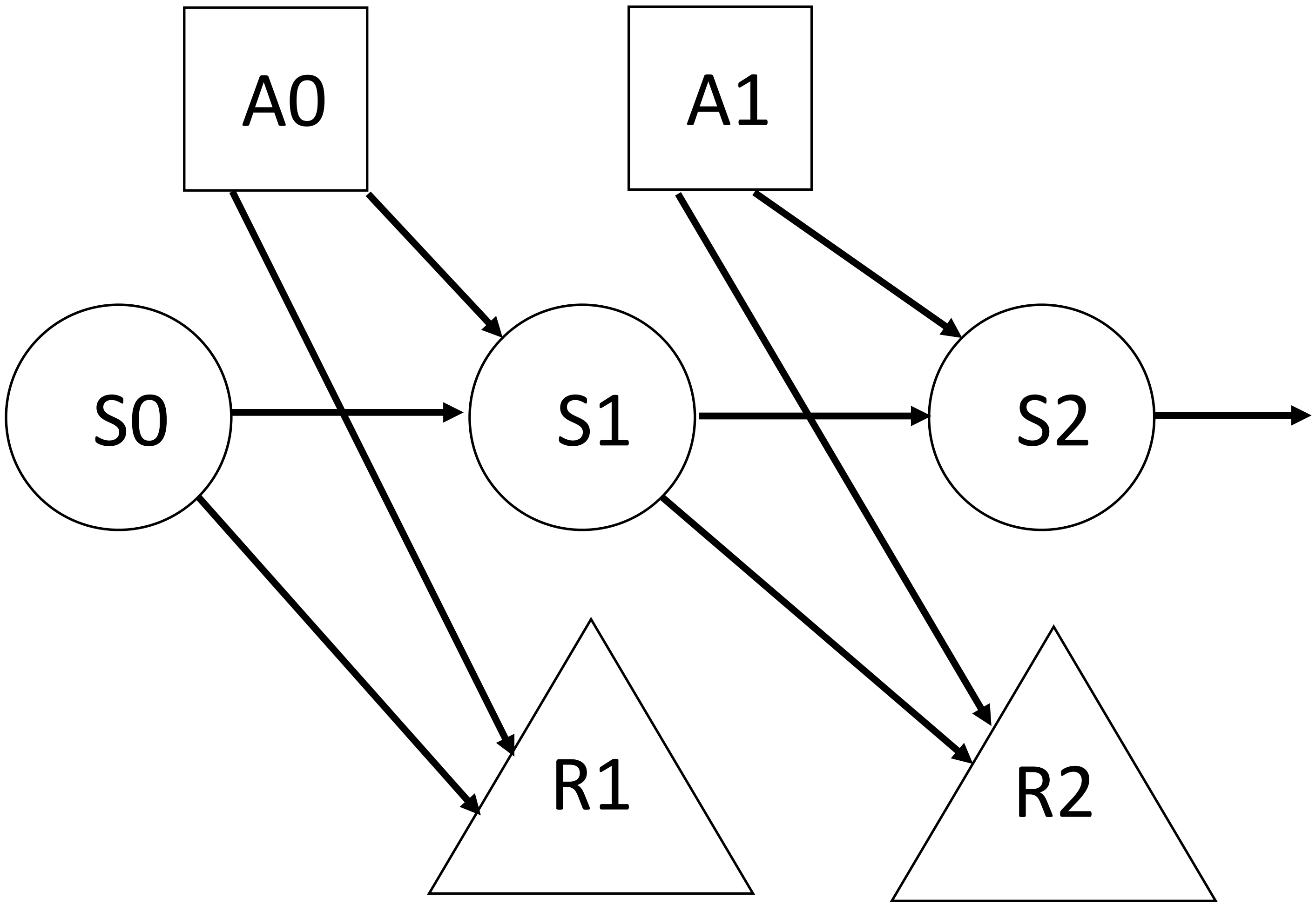
The agent and its environment



How can we formalize this mathematically?

Markov Decision Process (MDP)

- **Markov property:** current state completely characterizes state of the world.
- Defined by: (S, A, R, P, γ)
  - **S:** set of possible states
  - **A:** set of possible actions
  - **R:** reward for a given (state, action) pair
  - **$P(s_t | s_{t-1}, a_t)$:** transition probability
  - **γ:** Discount factor (usually close to 1)



Markov Decision Process (MDP)

- At time step t = 0, environment samples initial state $s_0 \sim P(s_0)$
- Then, for t = 0 until done:
  - Agent selects action a_t
  - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
  - Agent receives reward r_t and next state s_{t+1}.

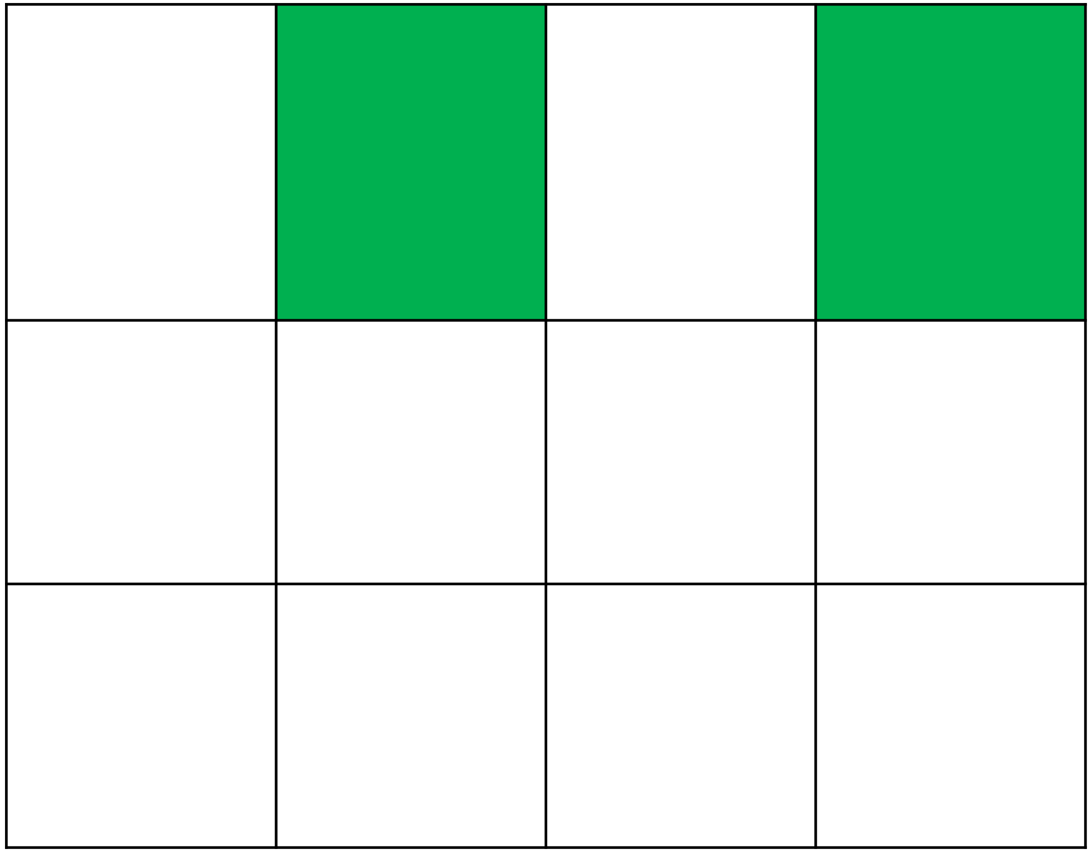
- A policy π is a function which specifies what action to take by the agent in each state.
- **Objective:** find a policy **π*** that maximizes cumulative discounted reward $\sum_{t \geq 0} \gamma^t r_t$

A simple MDP: Grid World

actions = {

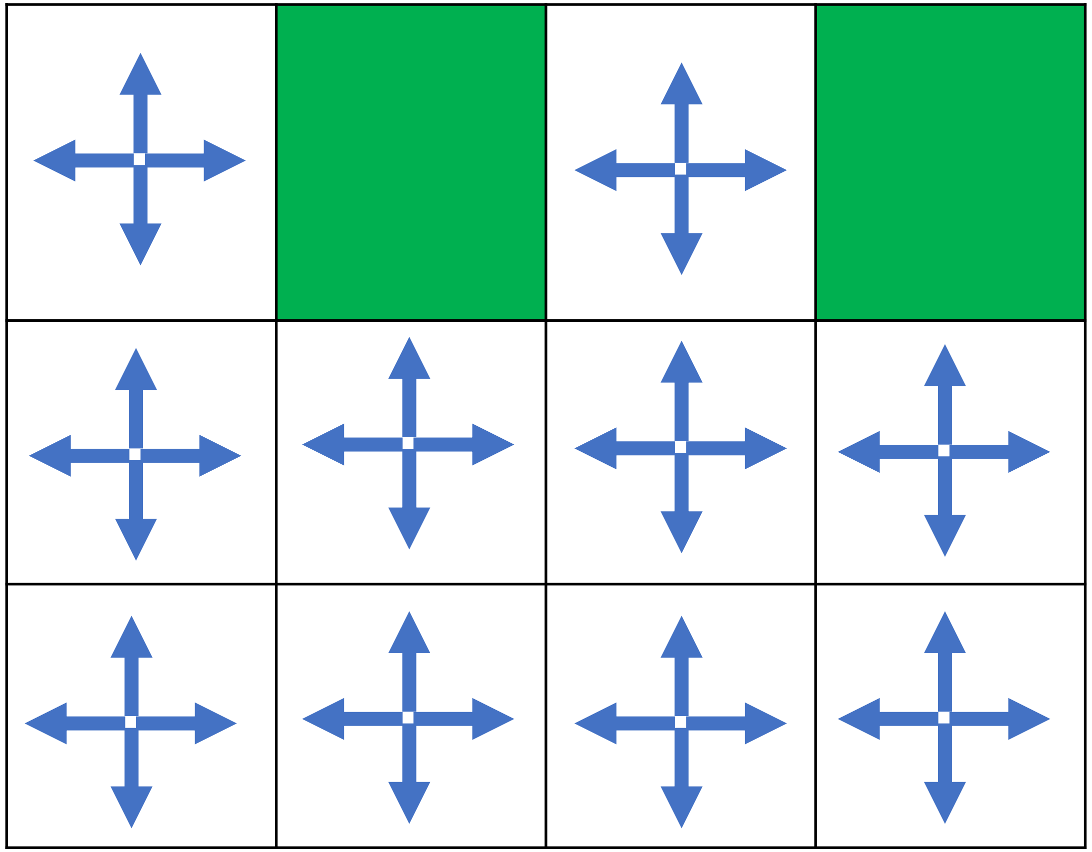
1. right
2. left
3. up
4. down

}

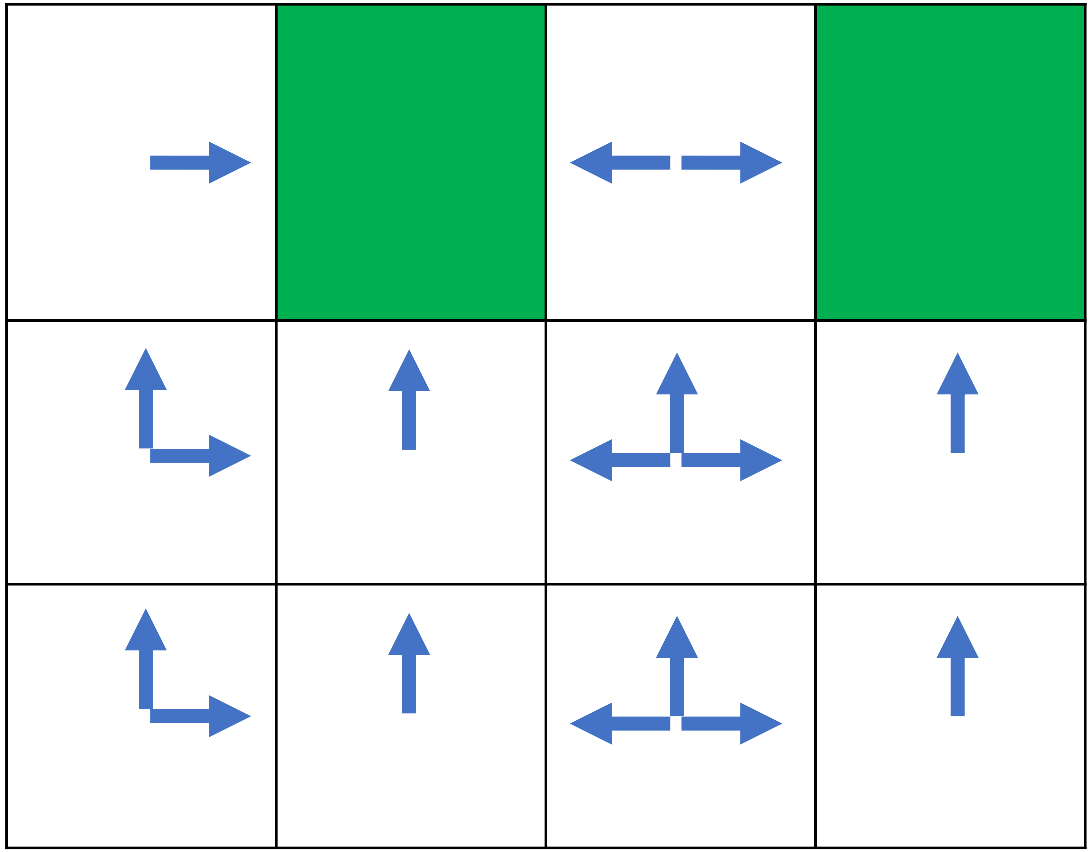


Objective: reach one of the terminal states (green) with the least number of actions

A simple MDP: Grid World



Random Policy



Optimal Policy

Definitions: Value function and Q-value function

- Following a policy produces sample trajectories (or paths) s_0, a_0, r_0, s_1, a_1, r_1, …

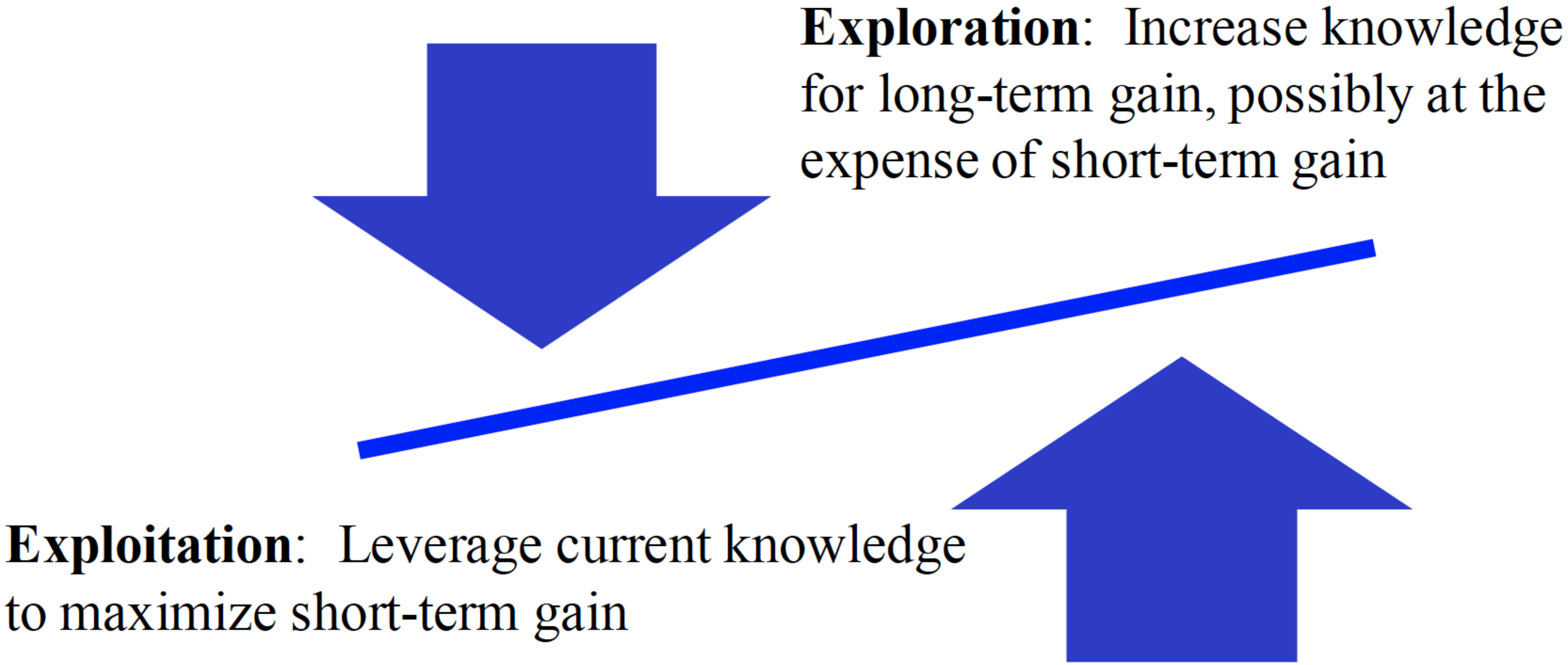
- **How good is a state?**
  - The **value function** at state s is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

- **How good is a state-action pair?**
  - The **Q-value function** at state s **and** action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Exploration vs Exploitation

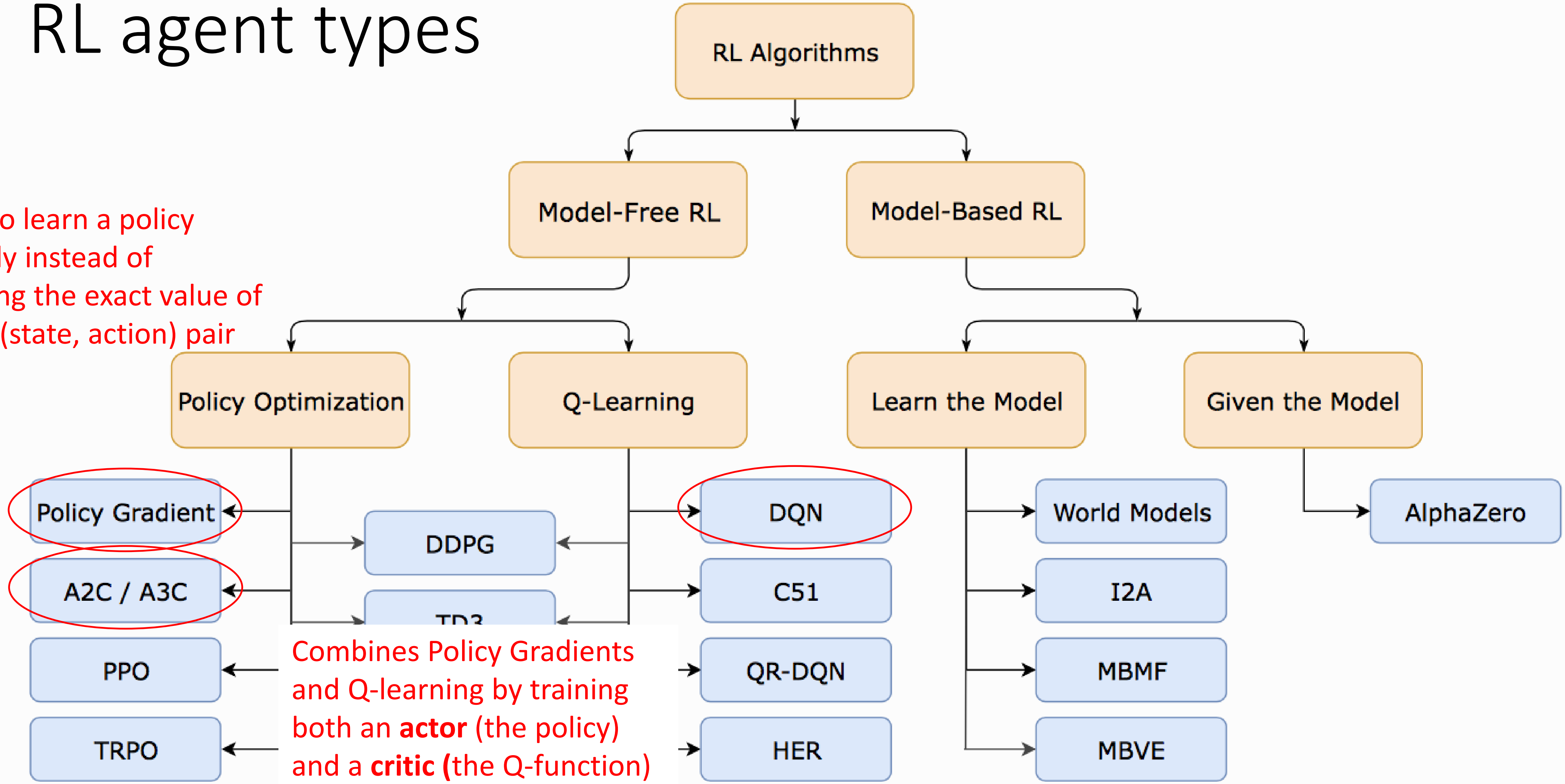


During training, we could e.g.:

30% of the time we choose a random action

70% of the time we choose an action with the most expected value

RL agent types



Tries to learn a policy directly instead of learning the exact value of every (state, action) pair

Combines Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function) -> 2 neural nets

Source: spinningup.openai.com

Summary

- In these 2 hours, we started from the basics (linear & logistic regression) and explored several models and learning paradigms.

- The last few years have seen a high growth in the take-up of ML by the particle accelerator and experimental physics community
  - Deep learning developments
  - Increase in scale and complexity of machines
  - Availability of “AI-ready” data

- ML will be a key tool to help meet demands for boosting performance, increasing autonomy and availability/reliability.

Back up slides

Neural Network backpropagation - details

Nomenclature

x Input training vector:

$$\mathbf{x} = (x_1, \ldots, x_i, \ldots, x_n).$$

t Output target vector:

$$\mathbf{t} = (t_1, \ldots, t_k, \ldots, t_m).$$

δ_k Portion of error correction weight adjustment for w_{jk} that is due to an error at output unit Y_k; also, the information about the error at unit Y_k that is propagated back to the hidden units that feed into unit Y_k.

δ_j Portion of error correction weight adjustment for v_{ij} that is due to the backpropagation of error information from the output layer to the hidden unit Z_j.

α Learning rate.

X_i Input unit i:
For an input unit, the input signal and output signal are the same, namely, x_i.

v_{0j} Bias on hidden unit j.

Z_j Hidden unit j:
The net input to Z_j is denoted z_in_j:

$$z\_in_j = v_{0j} + \sum_i x_i v_{ij}.$$

The output signal (activation) of Z_j is denoted z_j:

$$z_j = f(z\_in_j).$$

w_{0k} Bias on output unit k.

Y_k Output unit k:
The net input to Y_k is denoted y_in_k:

$$y\_in_k = w_{0k} + \sum_j z_j w_{jk}.$$

The output signal (activation) of Y_k is denoted y_k:

$$y_k = f(y\_in_k).$$

Training Algorithm

Step 0. Initialize weights.
(Set to small random values).

Step 1. While stopping condition is false, do Steps 2–9.

Step 2. For each training pair, do Steps 3–8.

Feedforward:

Step 3. Each input unit (X_i, $i = 1, \ldots, n$) receives input signal x_i and broadcasts this signal to all units in the layer above (the hidden units).

Step 4. Each hidden unit (Z_j, $j = 1, \ldots, p$) sums its weighted input signals,

$$z\_in_j = v_{0j} + \sum_{i=1}^{n} x_i v_{ij},$$

applies its activation function to compute its output signal,

$$z_j = f(z\_in_j),$$

and sends this signal to all units in the layer above (output units).

Step 5. Each output unit (Y_k, $k = 1, \ldots, m$) sums its weighted input signals,

$$y\_in_k = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and applies its activation function to compute its output signal,

$$y_k = f(y\_in_k).$$

Backpropagation of error:

Step 6. Each output unit (Y_k, $k = 1, \ldots, m$) receives a target pattern corresponding to the input training pattern, computes its error information term,

$$\delta_k = (t_k - y_k) f'(y\_in_k),$$

calculates its weight correction term (used to update w_{jk} later),

$$\Delta w_{jk} = \alpha \delta_k z_j,$$

calculates its bias correction term (used to update w_{0k} later),

$$\Delta w_{0k} = \alpha \delta_k,$$

and sends δ_k to units in the layer below.

Step 7. Each hidden unit ($Z_j, j = 1, \ldots, p$) sums its delta inputs (from units in the layer above),

$$\delta\_in_j = \sum_{k=1}^{m} \delta_k w_{jk},$$

multiplies by the derivative of its activation function to calculate its error information term,

$$\delta_j = \delta\_in_j \, f'(z\_in_j),$$

calculates its weight correction term (used to update v_{ij} later),

$$\Delta v_{ij} = \alpha \delta_j x_i,$$

and calculates its bias correction term (used to update v_{0j} later),

$$\Delta v_{0j} = \alpha \delta_j.$$

Update weights and biases:

Step 8. Each output unit ($Y_k, k = 1, \ldots, m$) updates its bias and weights ($j = 0, \ldots, p$):

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}.$$

Each hidden unit ($Z_j, j = 1, \ldots, p$) updates its bias and weights ($i = 0, \ldots, n$):

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}.$$

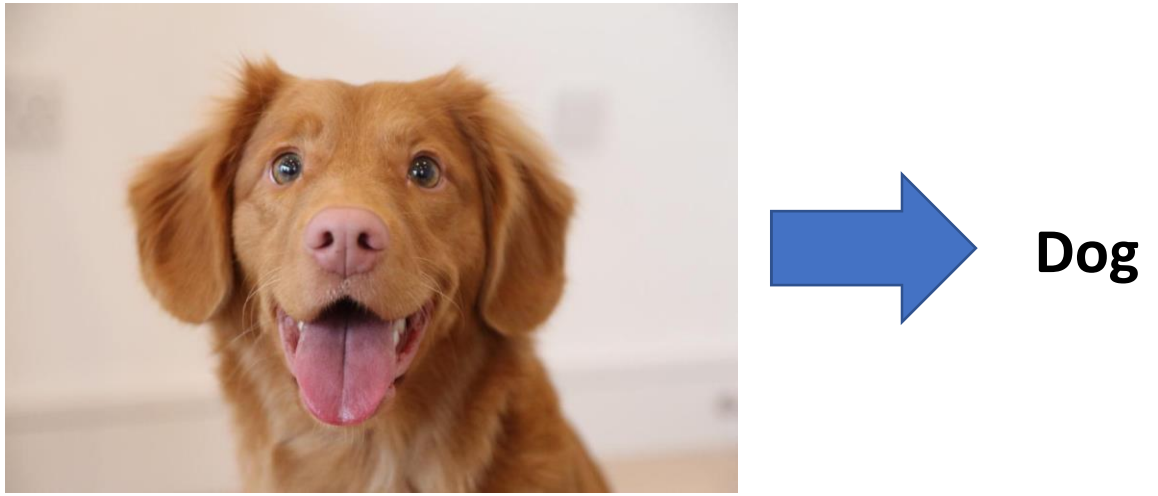
Step 9. Test stopping condition.

Reinforcement Learning

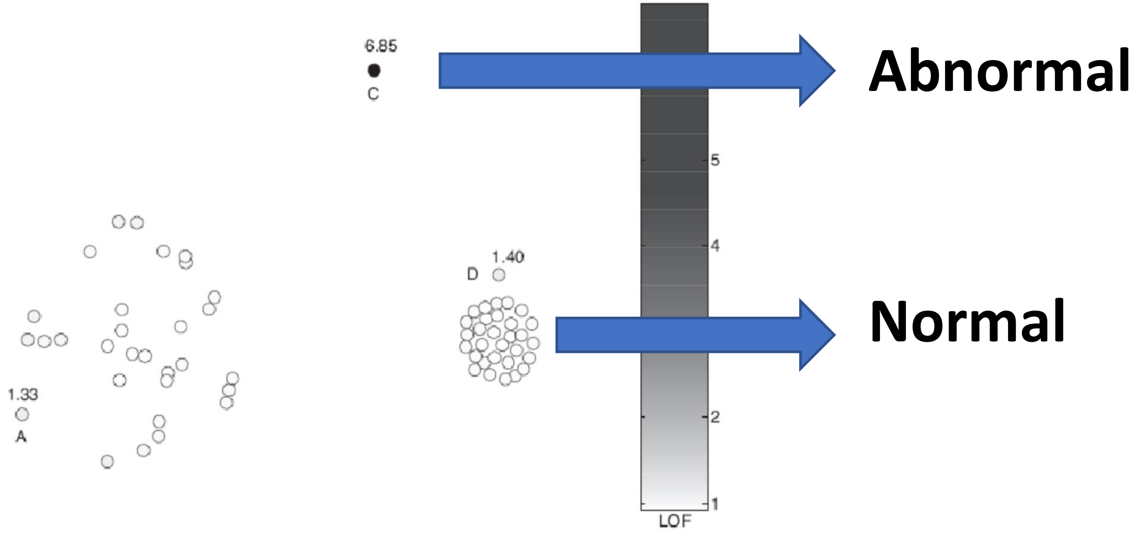
- What is Reinforcement Learning?
- RL terminology: states, actions, reward, policy
- Value function and Q-value function
- Q-learning and neural networks
- Grid World and Cart Pole

What is Reinforcement Learning?

- So far: **Supervised Learning**
  - **Data:** (X, y)
  - **Goal:** Learn a function to map X -> y
  - **Examples:** classification, regression, object detection etc



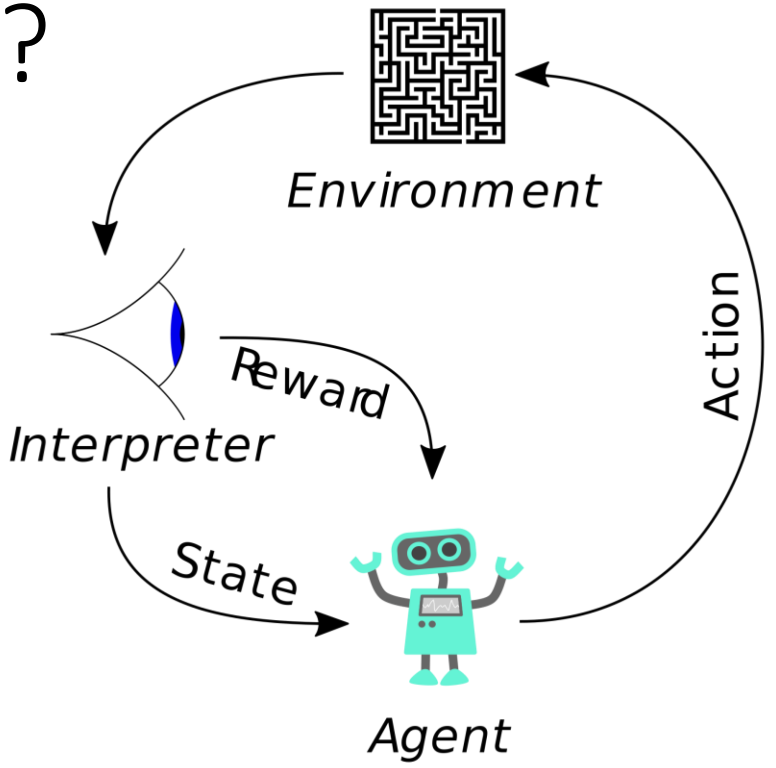
- So far: **Unsupervised Learning**
  - **Data**: X (no y)
  - **Goal:** Learn some underlying hidden structure in the data
  - **Examples:** clustering, dimensionality reduction, anomaly detection



relative density (LOF) outlier scores

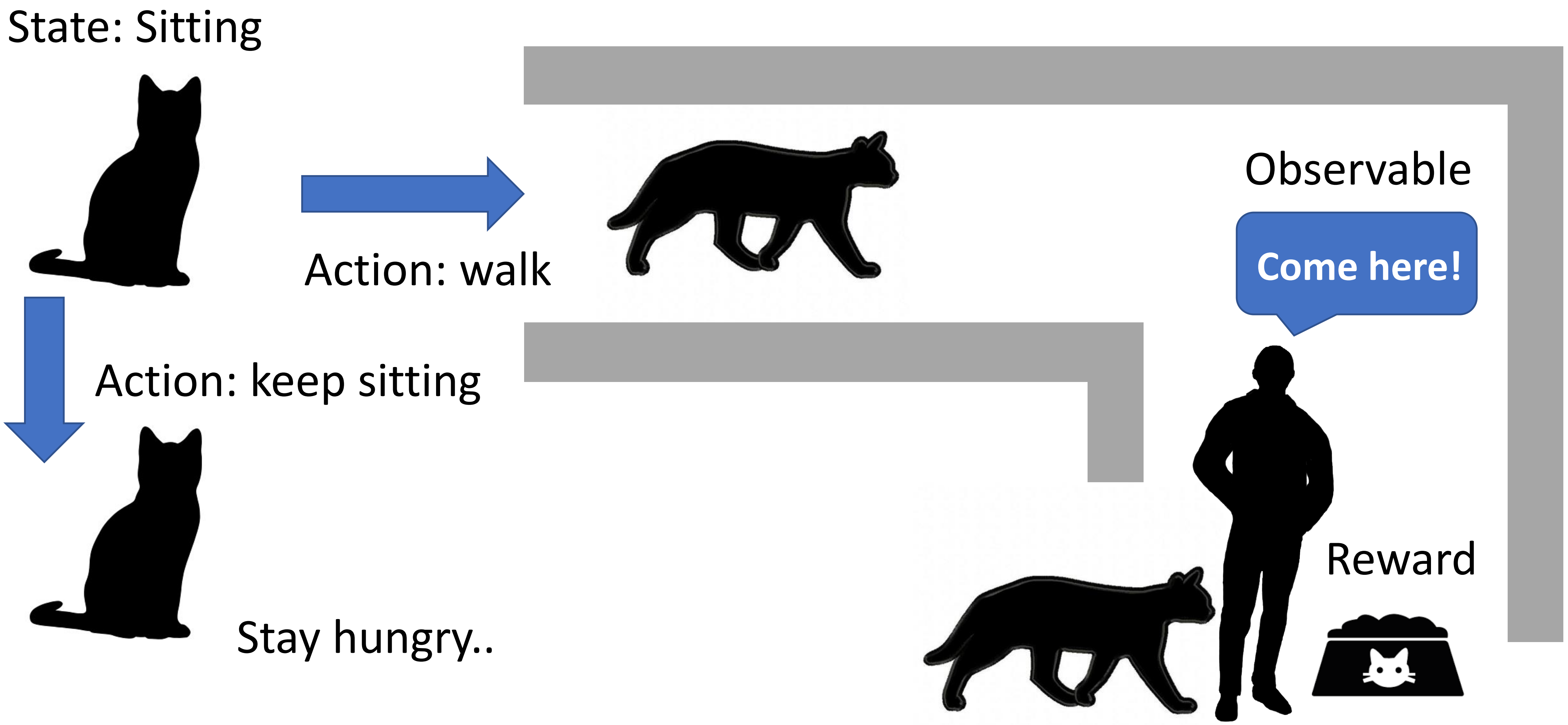
What is Reinforcement Learning?

- In Reinforcement Learning, an **agent** interacts with an **environment** to learn how to perform a particular task **well**.



- How is it different to the other learning paradigms?
  - There is no supervisor, only a **reward.**
  - The agent's actions **affect the subsequent data it receives**
  - **Feedback is delayed**, and may be received after several actions

Cat Agent


State: Sitting
Action: walk
Observable
Come here!
Action: keep sitting
Reward
Stay hungry..

Examples of Reinforcement Learning

Fly a helicopter

Make a robot walk

Manage an investment portfolio

Play Atari games better than humans

Rewards

- The agent receives feedback from the environment through reward
- A reward R_t is a scalar feedback signal
- It is an indication of how well the agent is doing at step t
- The agent's job is to **maximise cumulative reward**
- Examples:
  - Winning a game
  - Achieving design luminosity in a collider
  - Maintaining an inverted pendulum at the top

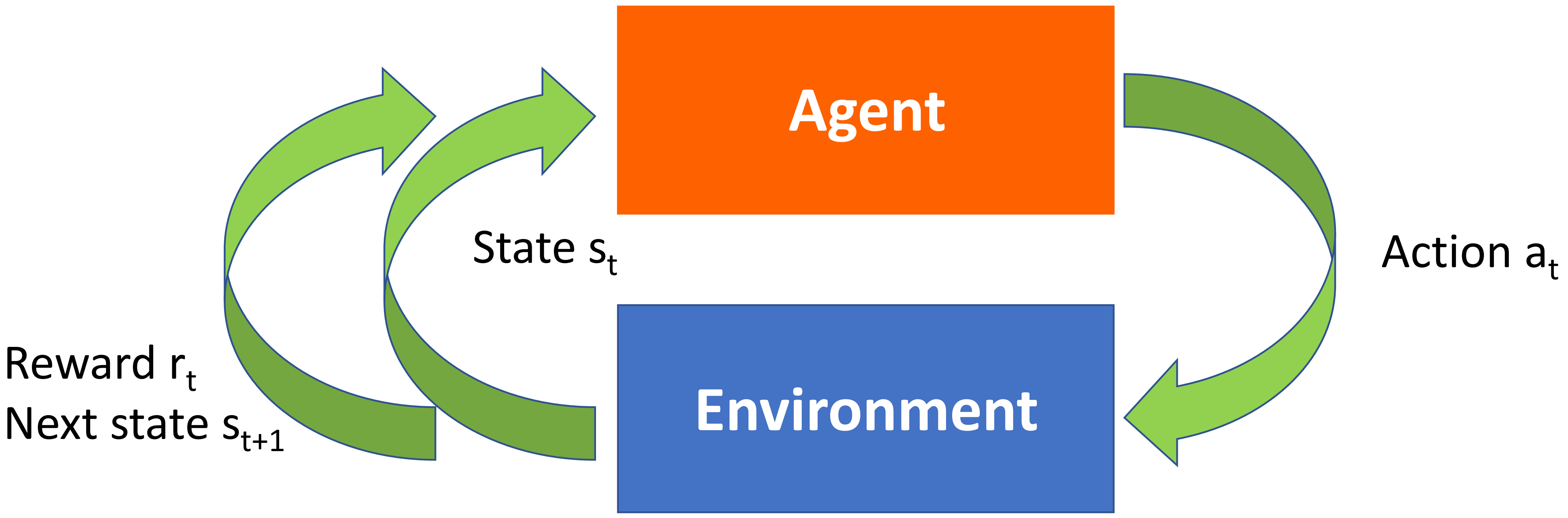
Sequential decision making

- **Goal:** select actions to maximise total future reward
- Actions may have long term consequences
- Reward may be delayed
- It may be better to sacrifice immediate reward to gain more long-term reward
- Examples:
  - A financial investment (may take months to mature)
  - Blocking opponent moves (might help winning probability many moves from now)

States

- **State:** what the agent is observing about the environment
- Examples:
  - Pixels in an image (of a game, of a driverless car, etc)
  - Data from beam instrumentation in an accelerator
  - The position of all pieces in a game of chess

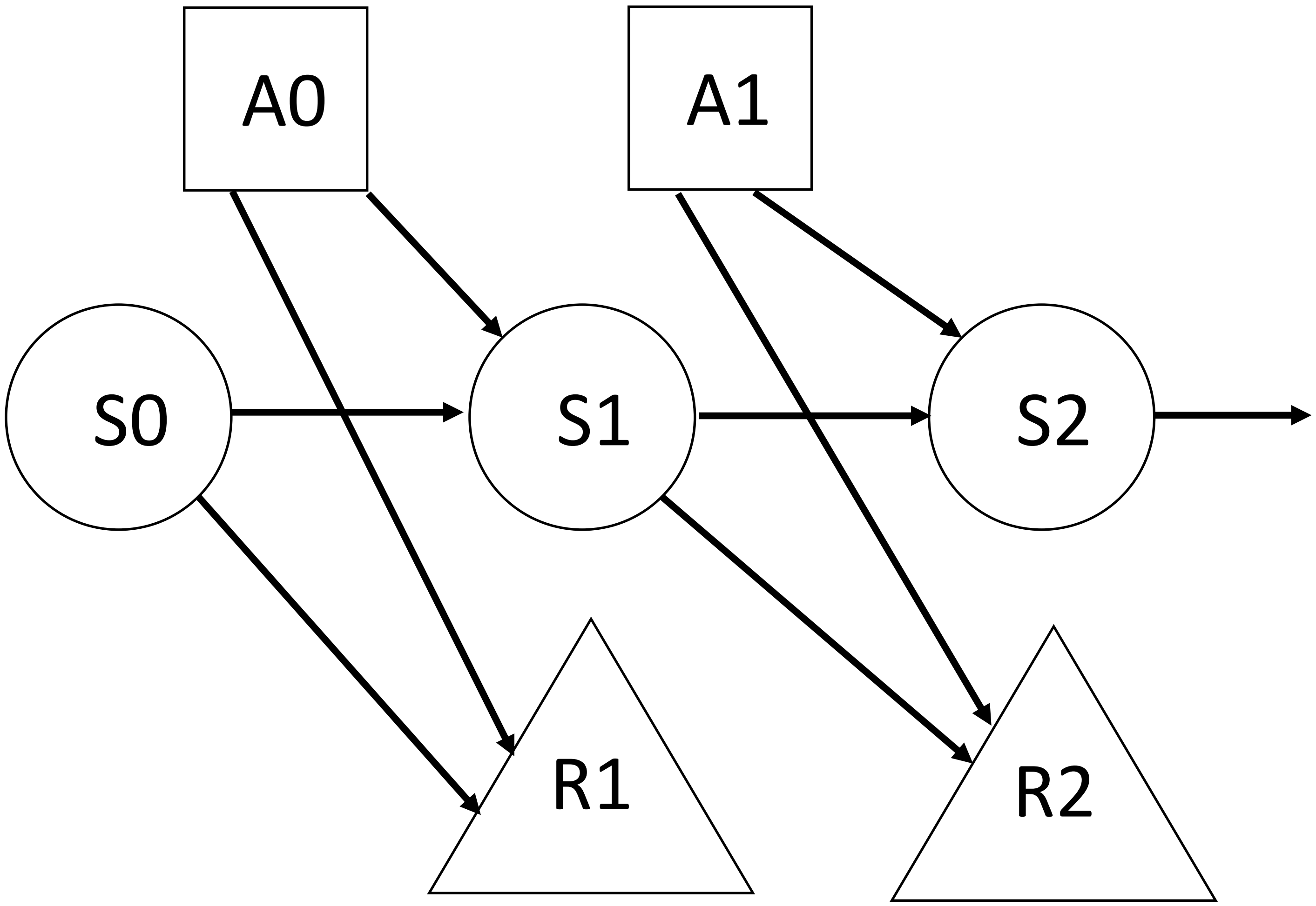
The agent and its environment



How can we formalize this mathematically?

Markov Decision Process (MDP)

- **Markov property:** current state completely characterizes state of the world.
- Defined by: (S, A, R, P, γ)
  - **S:** set of possible states
  - **A:** set of possible actions
  - **R:** reward for a given (state, action) pair
  - **$P(s_t | s_{t-1}, a_t)$:** transition probability
  - **γ:** Discount factor (usually close to 1)



Markov Decision Process (MDP)

- At time step t = 0, environment samples initial state $s_0 \sim P(s_0)$
- Then, for t = 0 until done:
  - Agent selects action a_t
  - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
  - Agent receives reward r_t and next state s_{t+1}.

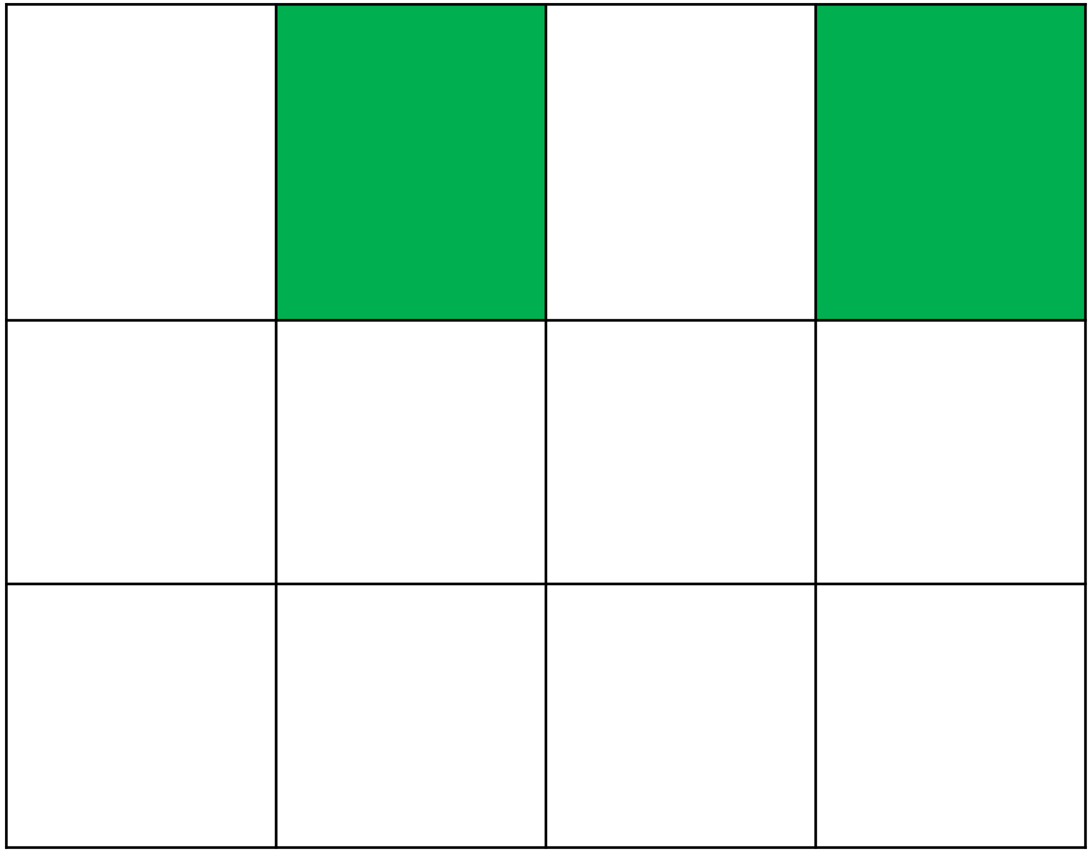
- A policy π is a function which specifies what action to take by the agent in each state.
- **Objective:** find a policy **π*** that maximizes cumulative discounted reward $\sum_{t \geq 0} \gamma^t r_t$

A simple MDP: Grid World

actions = {

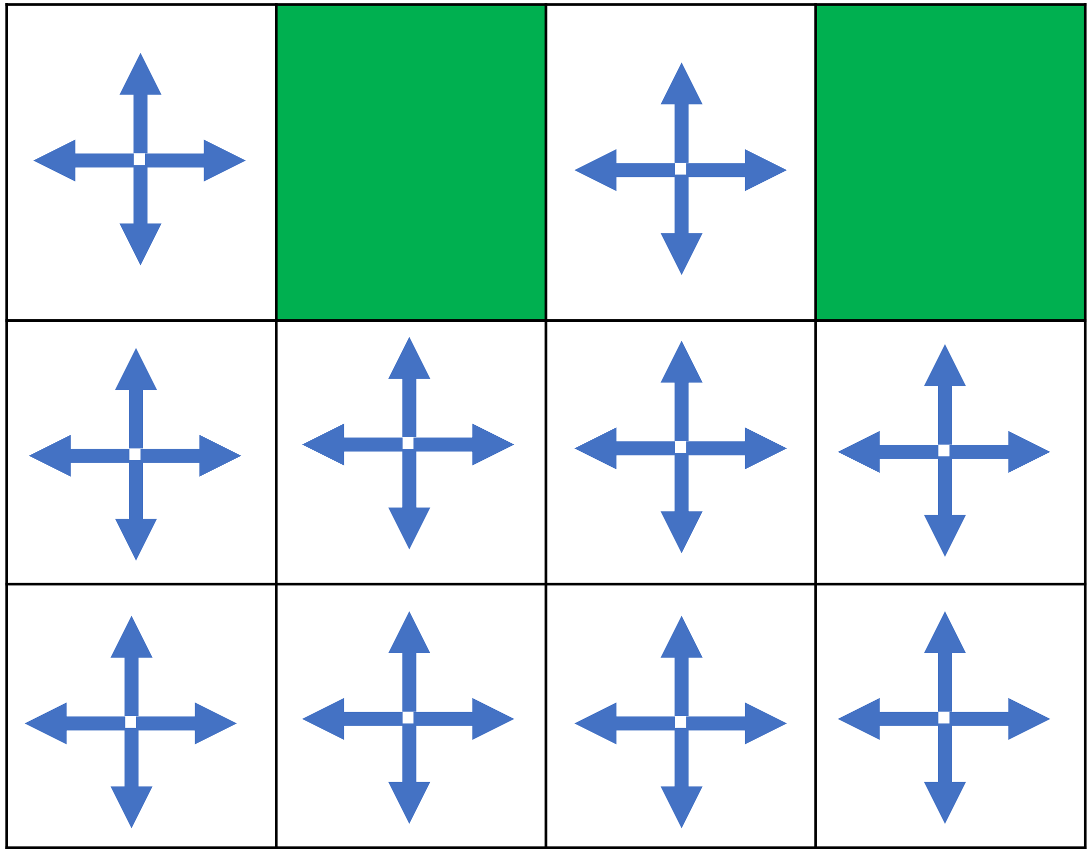
1. right
2. left
3. up
4. down

}

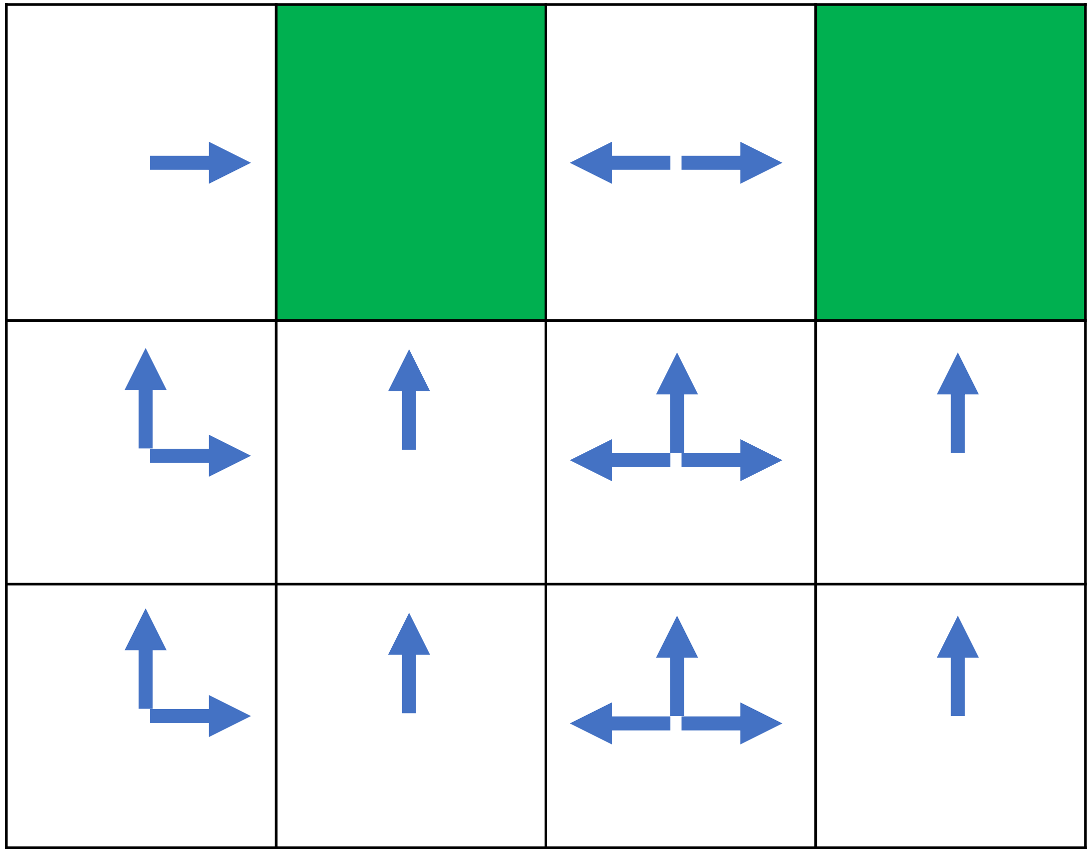


Objective: reach one of the terminal states (green) with the least number of actions

A simple MDP: Grid World



Random Policy



Optimal Policy

The optimal policy π*

- Need to find the optimal policy π* that maximizes the sum of rewards.
- To handle randomness (initial state, transition probability etc):
  - Maximize the **expected sum of rewards**

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right] \text{ with } s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$$

Definitions: Value function and Q-value function

- Following a policy produces sample trajectories (or paths) s_0, a_0, r_0, s_1, a_1, r_1, …

- **How good is a state?**
  - The **value function** at state s is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

- **How good is a state-action pair?**
  - The **Q-value function** at state s **and** action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Bellman equation

- The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_\pi \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

- Q* satisfies the **Bellman equation:**

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

- Intuition: if the optimal state-action values for the next time-step Q*(s',a') are known, then the optimal strategy is to take the action that maximizes the expected value of

$$r + \gamma Q^*(s',a')$$

- Optimal policy π* -> taking the best action in any state as specified by Q*.

Solving for the optimal policy

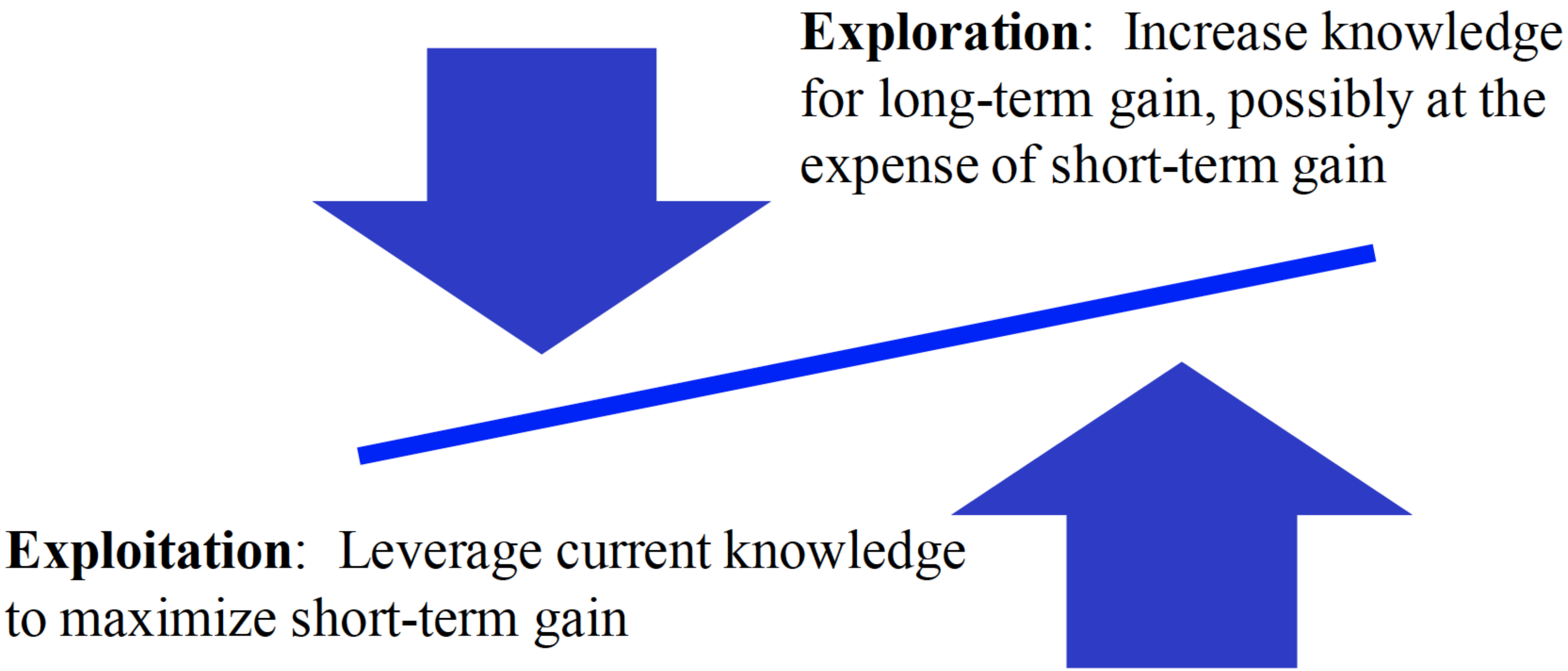
- **Value iteration algorithm:** use the Bellman equation as an iterative update:

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

- Q_i will converge to Q^* as $i \to$ infinity.

$$Q^{new}(s_t,a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Exploration vs Exploitation

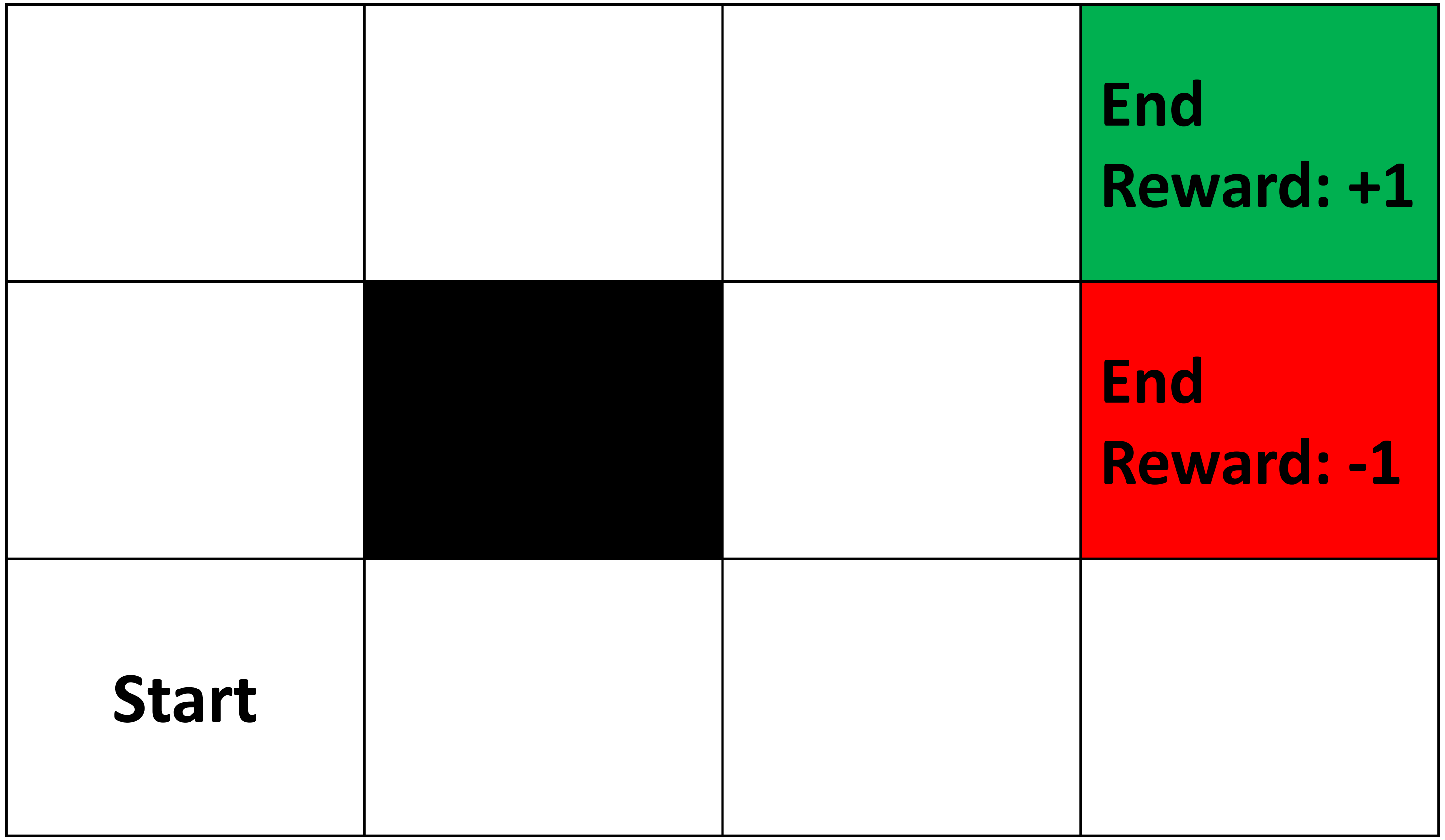


During training, we could e.g.:

30% of the time we choose a random action

70% of the time we choose an action with the most expected value

Grid world example



- Agent starts at bottom left.
- At each step, agent has 4 possible actions (up, down, left, right).
- Black square: agent cannot move through it.
- Assume each action is deterministic.

Grid world example

- First, define the grid world parameters:

```
import numpy as np

BOARD_ROWS = 3
BOARD_COLS = 4
WIN_STATE = (0, 3)
LOSE_STATE = (1, 3)
START = (2, 0)
#DETERMINISTIC = False
DETERMINISTIC = True
```

Grid world example

- Define the reward:

```
def giveReward(self):
    if self.state == WIN_STATE:
        return 1
    elif self.state == LOSE_STATE:
        return -1
    else:
        return 0
```

Grid world example

- Probabilistic result of taking an action:

```
def _chooseActionProb(self, action):
    if action == "up":
        return np.random.choice(["up", "left", "right"], p=[0.8, 0.1, 0.1])
    if action == "down":
        return np.random.choice(["down", "left", "right"], p=[0.8, 0.1, 0.1])
    if action == "left":
        return np.random.choice(["left", "up", "down"], p=[0.8, 0.1, 0.1])
    if action == "right":
        return np.random.choice(["right", "up", "down"], p=[0.8, 0.1, 0.1])
```

Grid world example

- Define how the state is updated when the action is taken by the agent.
- Need to check that the next state is not the black box or else outside the grid.

```
def nxtPosition(self, action):
    """
    action: up, down, left, right
    -------------
    0 | 1 | 2| 3|
    1 |
    2 |
    return next position on board
    """
    if self.determine:
        if action == "up":
            nxtState = (self.state[0] - 1, self.state[1])
        elif action == "down":
            nxtState = (self.state[0] + 1, self.state[1])
        elif action == "left":
            nxtState = (self.state[0], self.state[1] - 1)
        else:
            nxtState = (self.state[0], self.state[1] + 1)
        self.determine = False
    else:
        # non-deterministic
        action = self._chooseActionProb(action)
        self.determine = True
        nxtState = self.nxtPosition(action)

    #self.showBoard()

    # if next state is legal
    if (nxtState[0] >= 0) and (nxtState[0] <= 2):
        if (nxtState[1] >= 0) and (nxtState[1] <= 3):
            if nxtState != (1, 1):
                return nxtState
    return self.state
```

Grid world example

- Tradeoff between exploration (new info) and exploitation (greedy actions):

```
def chooseAction(self):
    # choose action with most expected value
    mx_nxt_reward = 0
    action = ""

    if np.random.uniform(0, 1) <= self.exp_rate:
        action = np.random.choice(self.actions)
    else:
        # greedy action
        for a in self.actions:
            current_position = self.State.state
            nxt_reward = self.Q_values[current_position][a]
            if nxt_reward >= mx_nxt_reward:
                action = a
                mx_nxt_reward = nxt_reward
        # print("current pos: {}, greedy aciton: {}".format(self.State.state, action))

    if action == "":
        action = np.random.choice(self.actions)

    return action
```

Grid world example

- Define stopping condition:

```
def isEndFunc(self):
    if (self.state == WIN_STATE) or (self.state == LOSE_STATE):
        self.isEnd = True
```

Grid world example

- Bring everything together:

```
def play(self, rounds=10):
    i = 0
    while i < rounds:
        # to the end of game back propagate reward
        if self.State.isEnd:
            # back propagate
            reward = self.State.giveReward()
            for a in self.actions:
                self.Q_values[self.State.state][a] = reward
            print("Game End Reward", reward)
            for s in reversed(self.states):
                current_q_value = self.Q_values[s[0]][s[1]]
                reward = current_q_value + self.lr * (self.decay_gamma * reward - current_q_value)
                self.Q_values[s[0]][s[1]] = round(reward, 3)
            self.reset()
            i += 1
        else:
            action = self.chooseAction()

            # append trace
            self.states.append([(self.State.state), action])
            print("current position {} action {}".format(self.State.state, action))
            # by taking the action, it reaches the next state
            self.State = self.takeAction(action)
            # mark is end
            self.State.isEndFunc()
            print("nxt state", self.State.state)
            print("---------------------")
            self.isEnd = self.State.isEnd
```

Solving for the optimal policy: Q-learning

- **Value iteration algorithm:** use the Bellman equation as an iterative update:

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

- Q_i will converge to Q* as i -> infinity.

- What is the problem with this?
  - Not scalable: must compute Q(s, a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

- Solution: use a function approximator to estimate Q(s,a).
  - A **neural network!**

Solving for the optimal policy: Q-learning

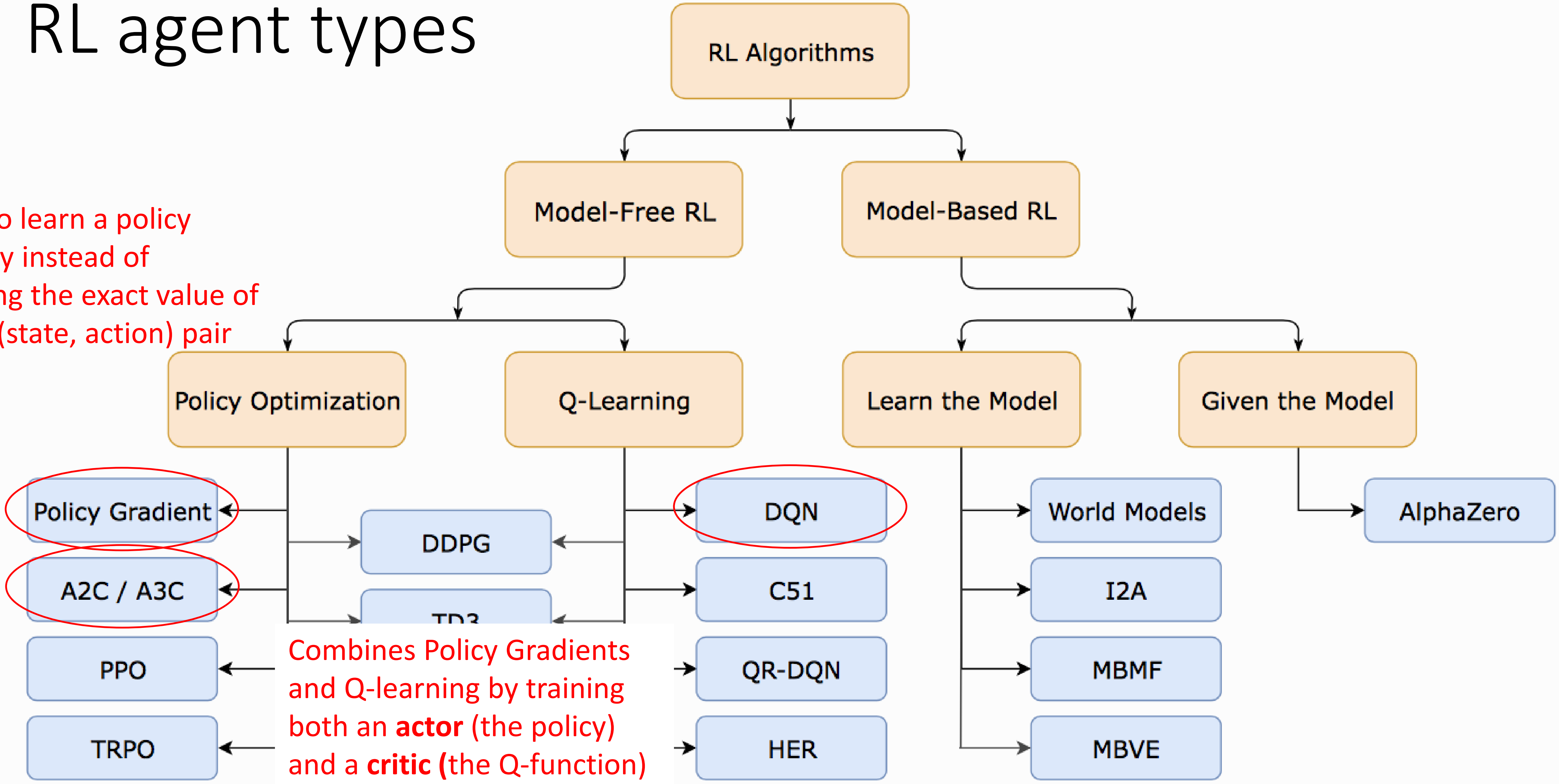
- Q-learning: use a function approximator to estimate the action-value function:

$$Q(s, a; \Theta) \approx Q^*(s, a)$$

Where Θ are the neural network weights which need to be learned.

- If the function approximator is a deep neural network -> **deep q-learning (DQN)!**

RL agent types

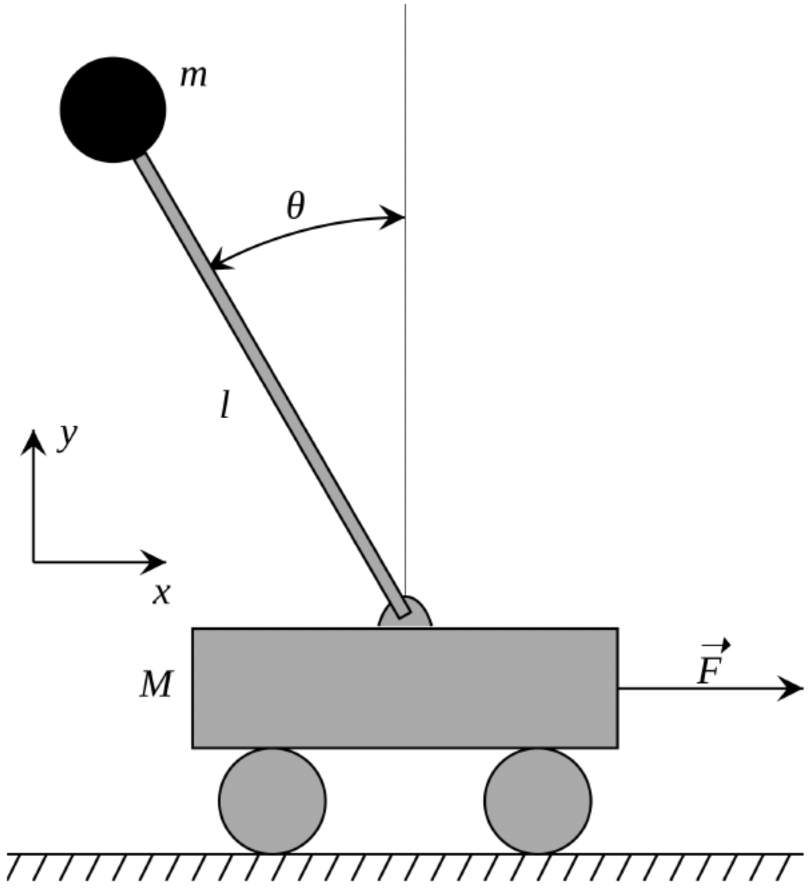


Tries to learn a policy directly instead of learning the exact value of every (state, action) pair

Combines Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function) -> 2 neural nets

Source: spinningup.openai.com

Cartpole Problem



- **Objective:** Balance a pole on top of a movable cart

- **State:** angle, angular speed, position, horizontal velocity
- **Action:** horizontal force applied on the cart (or not)
- **Reward:** +1 at each time step if the pole is upright (within some limits)

OpenAI Gym

- In order to train an agent to perform a task, we need a suitable physical environment.

- OpenAI gym provides a number of ready environments for common problems, e.g. Cart Pole, Atari Games, Mountain Car

- However, you can also define your own environment following the OpenAI Gym framework (e.g. physical model of accelerator operation)

OpenAI

OpenAI Gym – Cart Pole Environment

- Let's have a look at the Cart Pole environment in cartpole.ipynb
- Main component: **step function**
  - Updates state
  - Calculates reward
- Also has rendering functionality

Implementation of a DQN agent

- There are several ready implementations of RL agents
  - E.g. Keras RL

- We first define the Q network architecture (in Keras fashion):

```
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())
```

Implementation of a DQN agent

- We can use a ready-made policy (BoltzmannQPolicy)
  - Builds a probability law on q-values and returns an action selected randomly according to this law.

- We also define the number of actions, the learning rate and the number of steps that we want to train the agent for, trying to optimize some metric.

- Memory: stores the agent's experiences
- Number of warmup steps: avoids early overfitting
- Target Model update: how often are weights of target network updated

```
memory = SequentialMemory(limit=50000, window_length=1)
policy = BoltzmannQPolicy()
dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory, nb_steps_warmup=10,
               target_model_update=1e-2, policy=policy)
dqn.compile(Adam(lr=1e-3), metrics=['mae'])

history = dqn.fit(env, nb_steps=100, visualize=True, verbose=2)
```