ECS171: Machine Learning

L16: Unsupervised Learning III and reinforcement learning

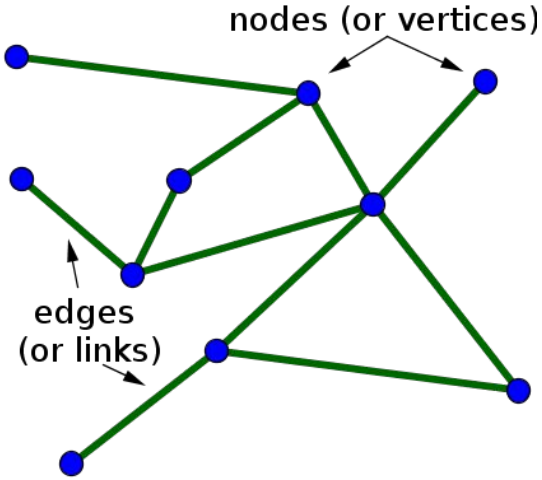
Instructor: Prof. Maike Sonnewald
TAs: Pu Sun & Devashree Kataria

Intended Learning Outcomes

- Describe the use of graphs in unsupervised learning
- Describe example of a k-NN graph
  - Computational cost implications
  - Concept of k
- Describe and appreciate the application of geometry to learning UMAP example
  - Role of weights in graph
  - Parameters n_neighbours and min_dist roles in simple examples
- Describe and compare/contrast reinforcement learning vrt supervised and unsupervised learning
  - Describe concepts of state, action, revard, policy and value in simple example like tic, tac, toe
  - Markov Decision Problem
  - Be familiar with Q-learning

Graphs

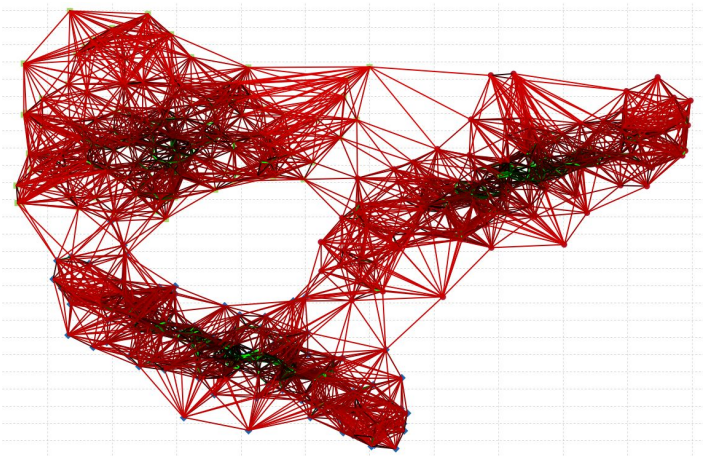
- Very powerful method to find patterns in data
- Can be based on a similarity or dissimilarity measure
- Non-parametric
- A vertex for each point, and an edge from *p* to *q*
  - Whenever *q* is a nearest neighbor of *p*: a point whose distance from *p* is minimum among all the given points other than *p* itself
- Can be used to understand the topology of a dataset

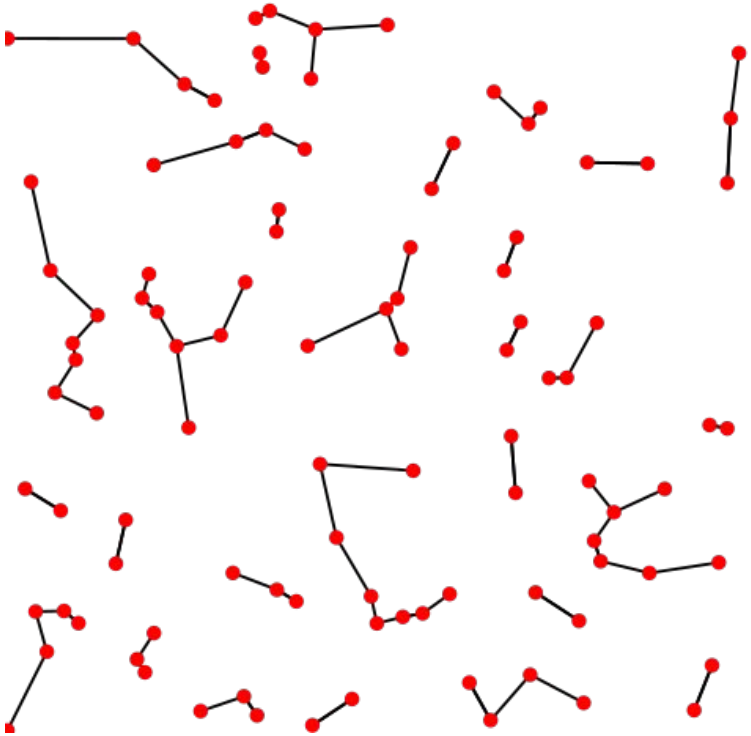


k-Nearest Neighbor graph

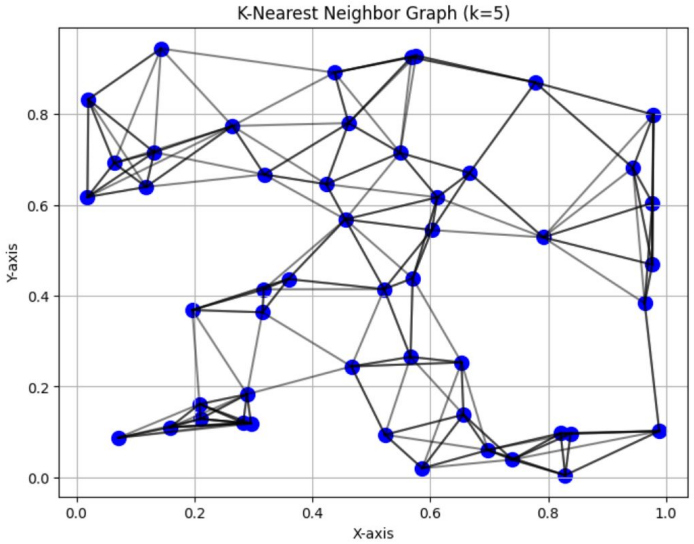
- A graph in which two vertices *p* and *q* are connected by an edge
    - Condition: distance between *p* and *q* is among the *k*-th smallest distances from *p* to other objects from *P*.
- Put differently: it is a graph where each node is connected to the *k most similar entities*, as an attempt to insert some bias about what a good graph structure should look like

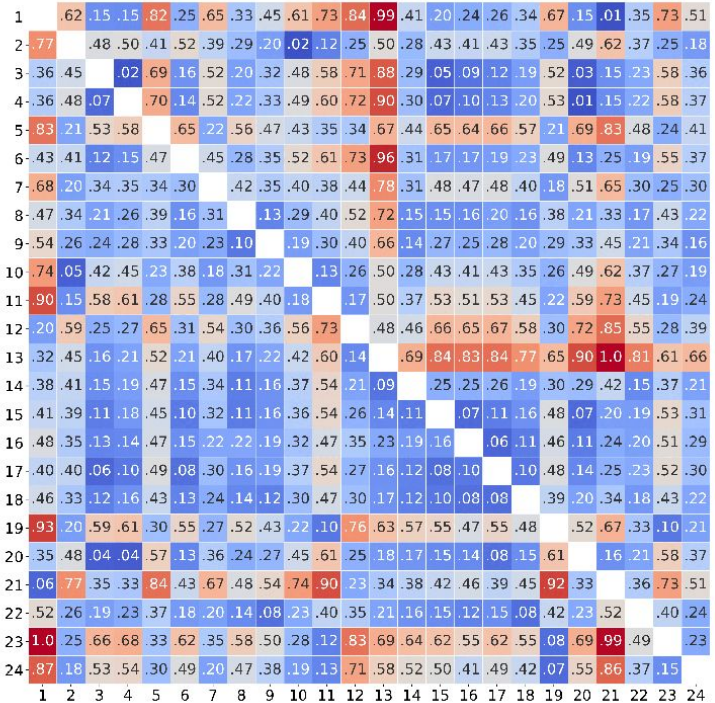
Setting k allows looking at global vs local structure





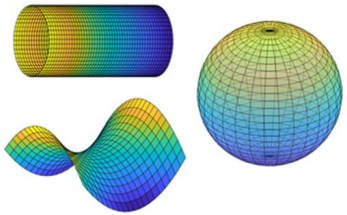
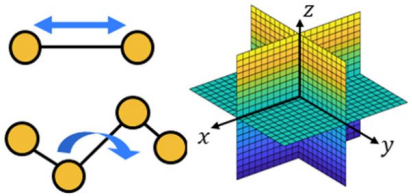
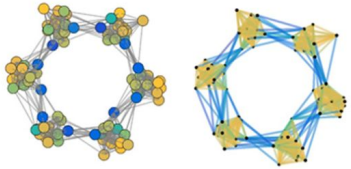
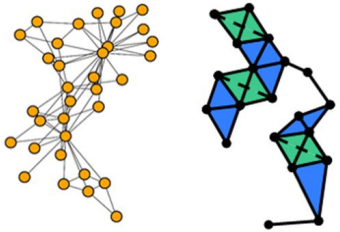
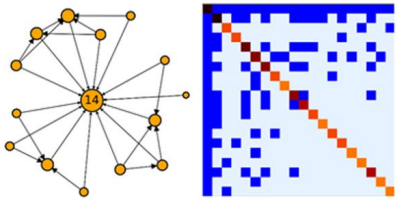
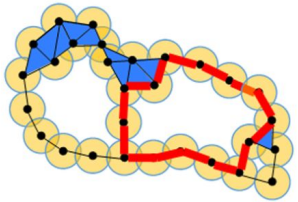
Pairwise distance matrix between points





Nobel et al., 2021

Topology and data in ML

	Representations	General descriptors	Intrinsic descriptors
Geometry	*1D curve, 2D surface, 3D volume, high-dimension geometry, etc.* 	*Position, dimension, distance, angle, perimeter, surface area, volume, etc.* 	*Gaussian curvature, Ricci curvature, Riemann curvature tensor, etc.* 
Topology	*Point cloud, curve, graph, network, simplicial complex, hypergraph, etc.* 	*Node degree, cluster coefficient, closeness, centrality, betweenness, etc* 	*Homotopy, homology, cohomology, Betti number, Euler characteristics, knot polynomial, Conley index, etc* 

Learning more/dimensionality reduction with graphs

Going from high to low dimensions can be done with graphs

Avoids making assumptions of e.g. orthogonality as in PCA

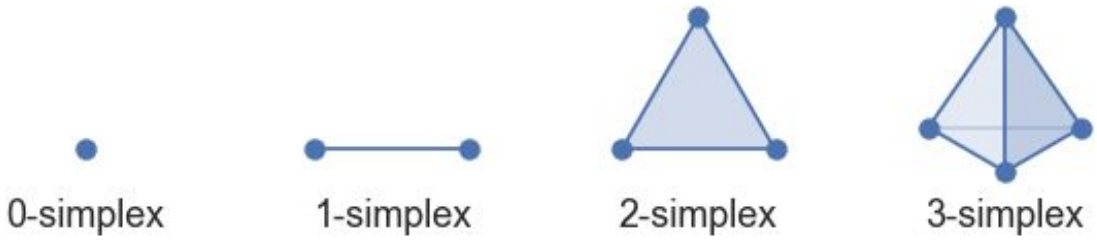
Use distances to determine and preserve the ‘latent space’ in fewer dimensions

Graph layouts

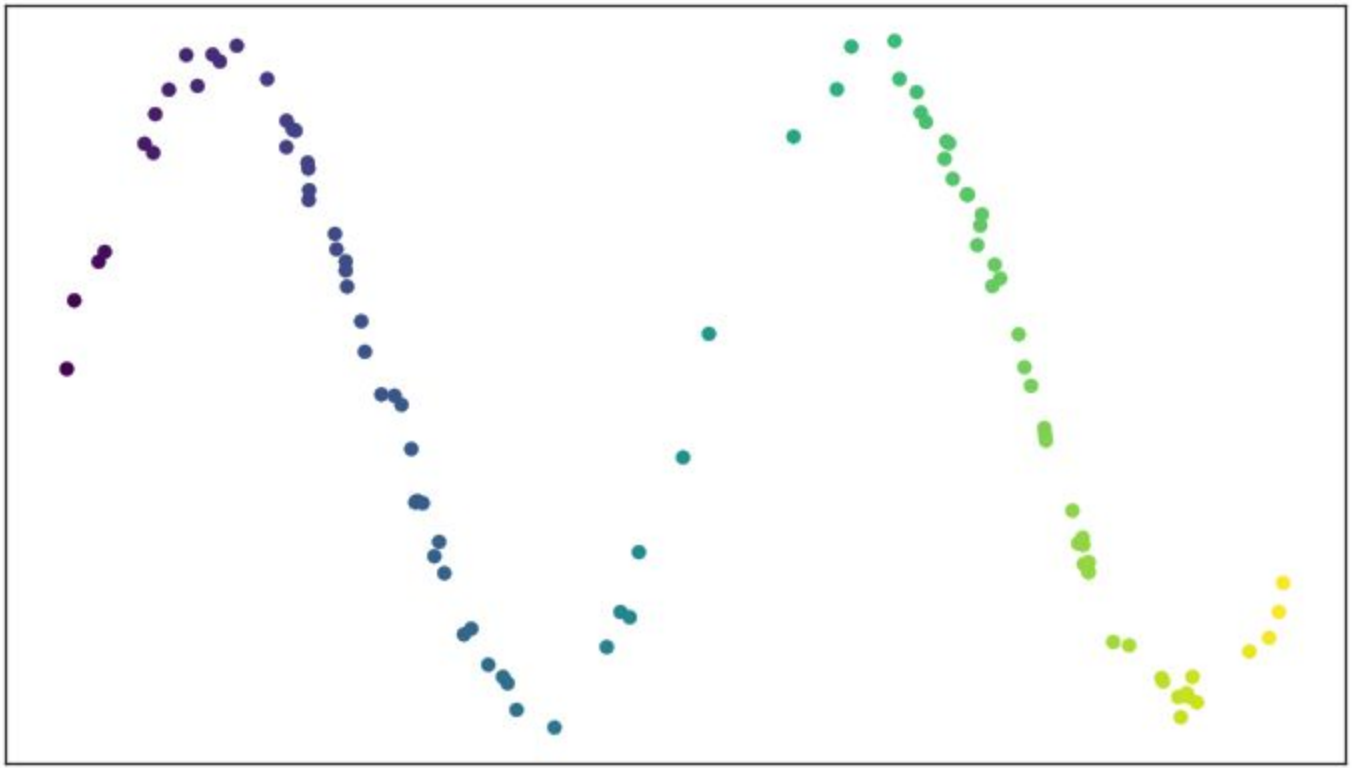
We have a simple graph if we only care about the 0- and 1-simplices

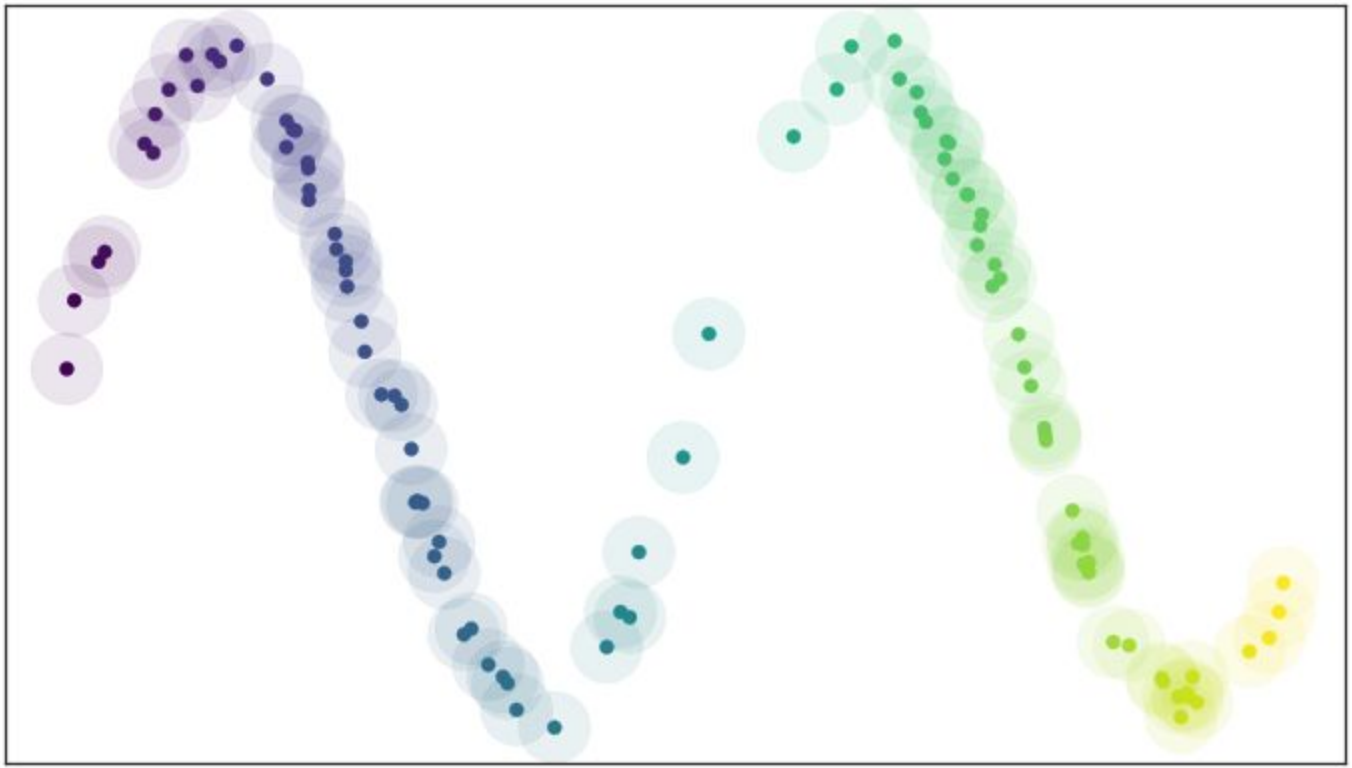
FInding the topological representation is just a graph

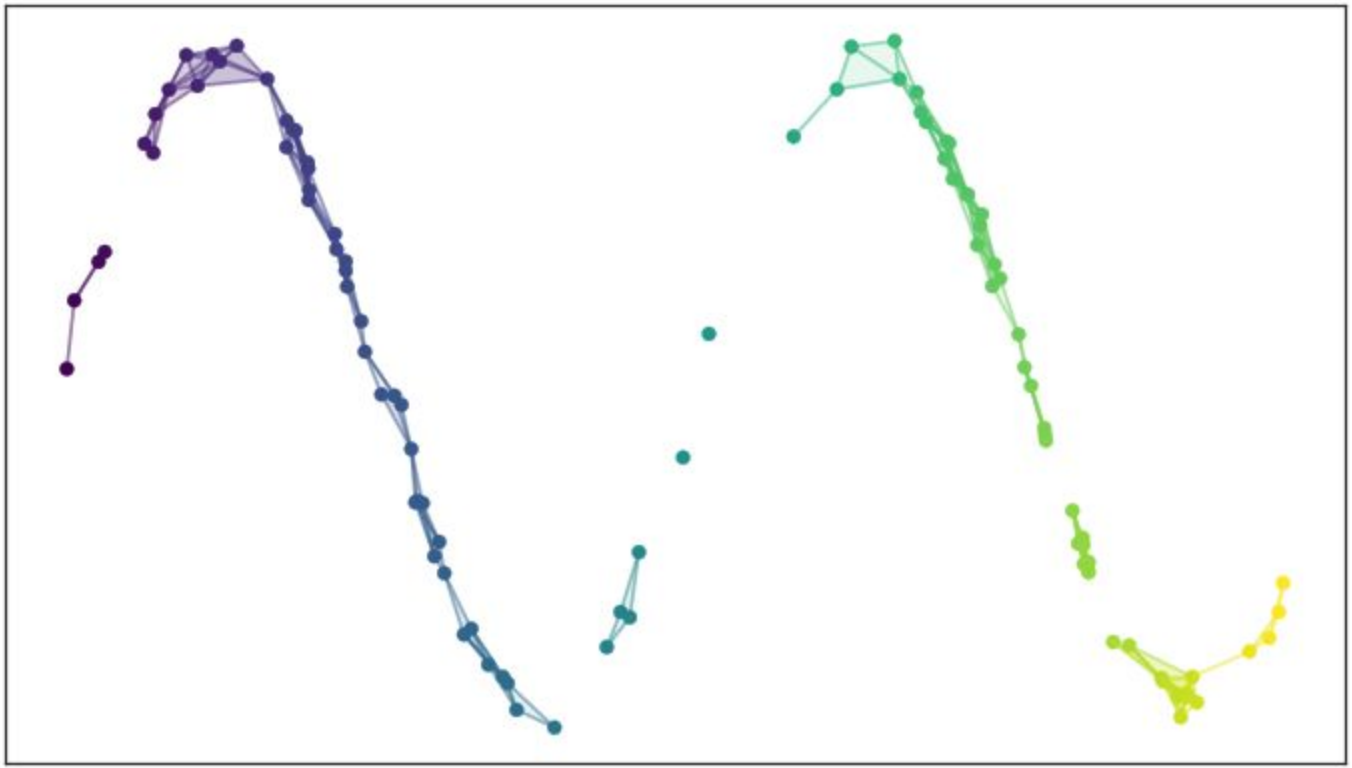
Finding a low dimensional representation can be described as a graph layout problem.



More details on what follows:
https://umap-learn.readthedocs.io/en/latest/how_umap_works.html#id6







Uniform Manifold Approximation Projection (UMAP)

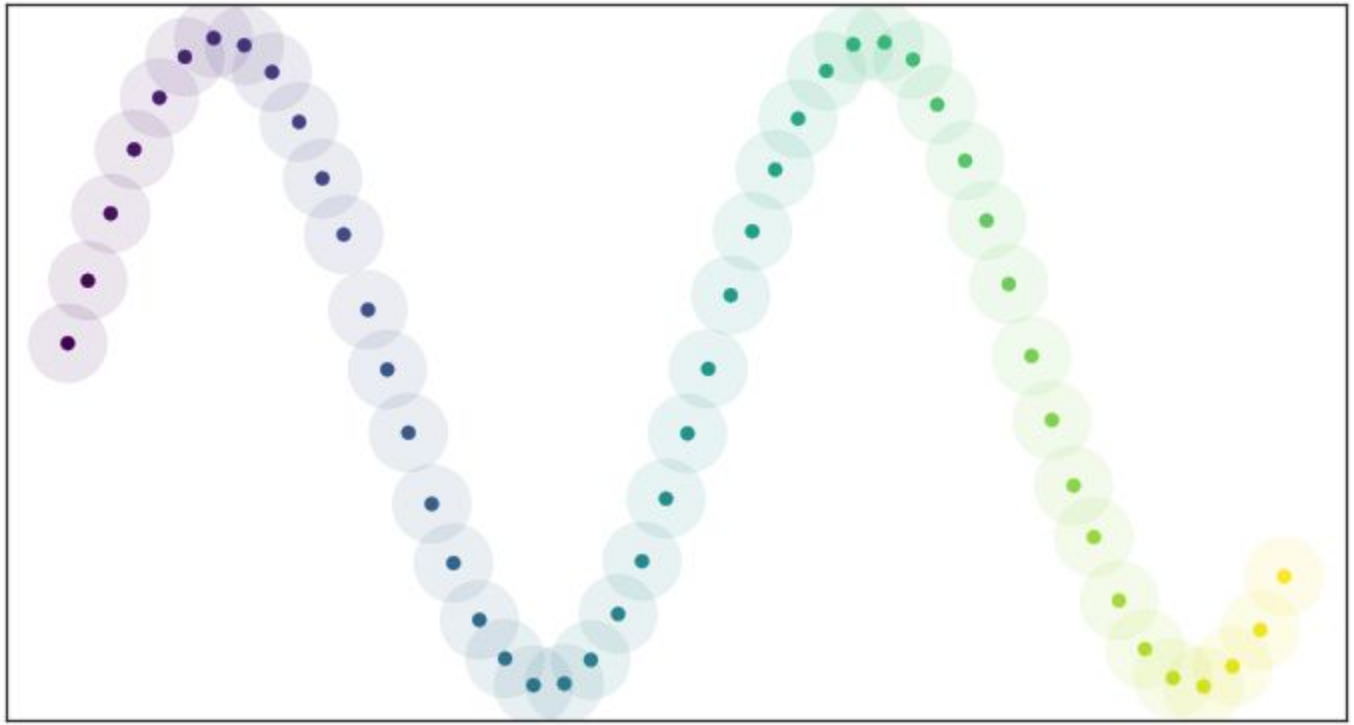
FInding a low-dimensional representation using weighted graphs as an optimisation problem

Desirable properties:

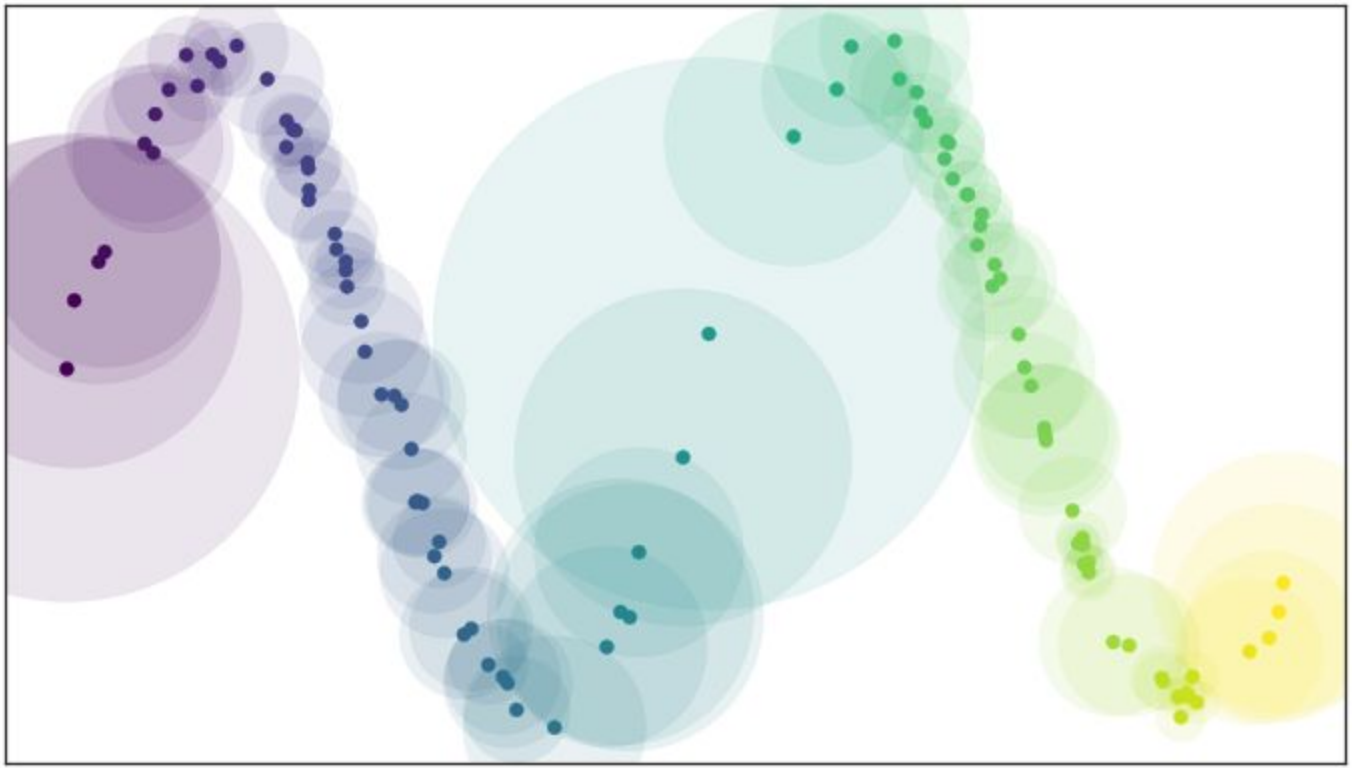
- Lower dimensions for visualisation
- Weights assure that ‘related’ data-points stay together

Assume data is uniformly distributed

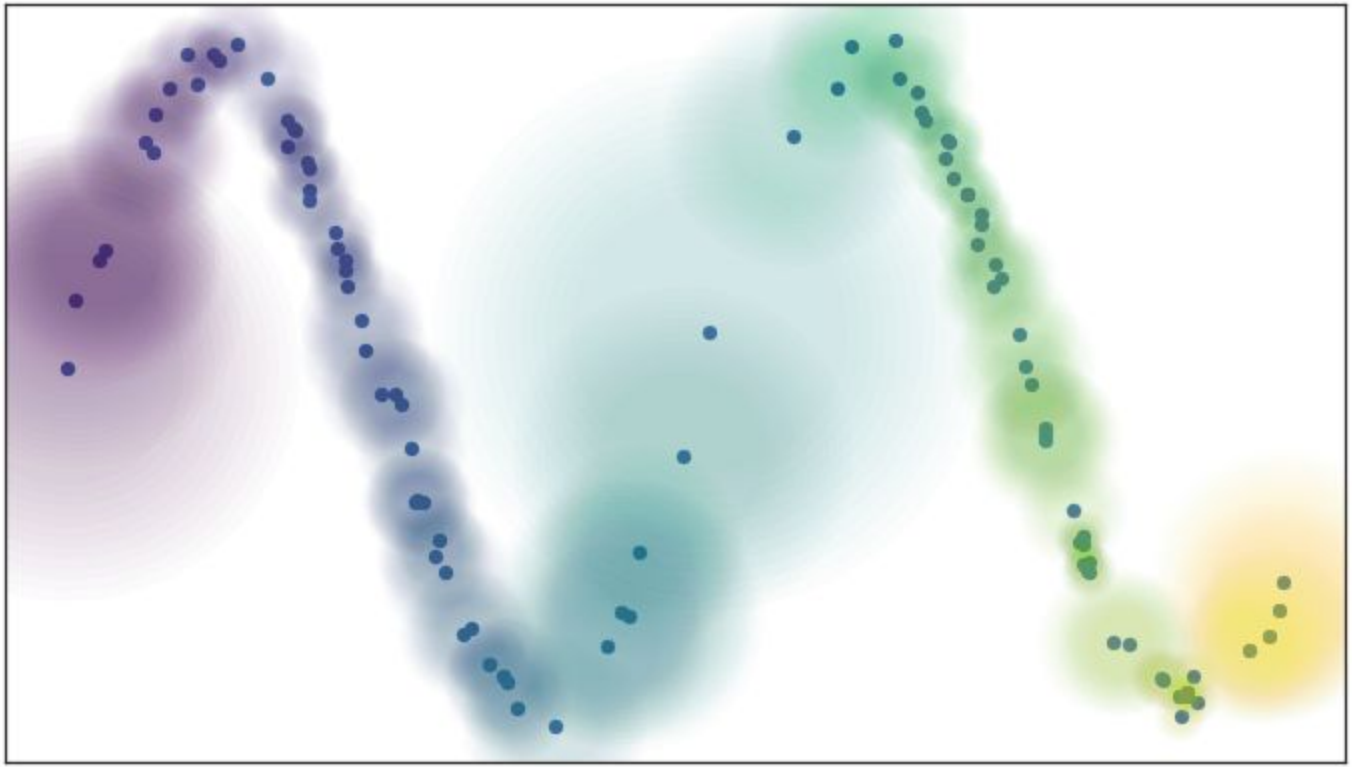
- Compute an approximation of local distance to each point
- Riemannian *geometry*



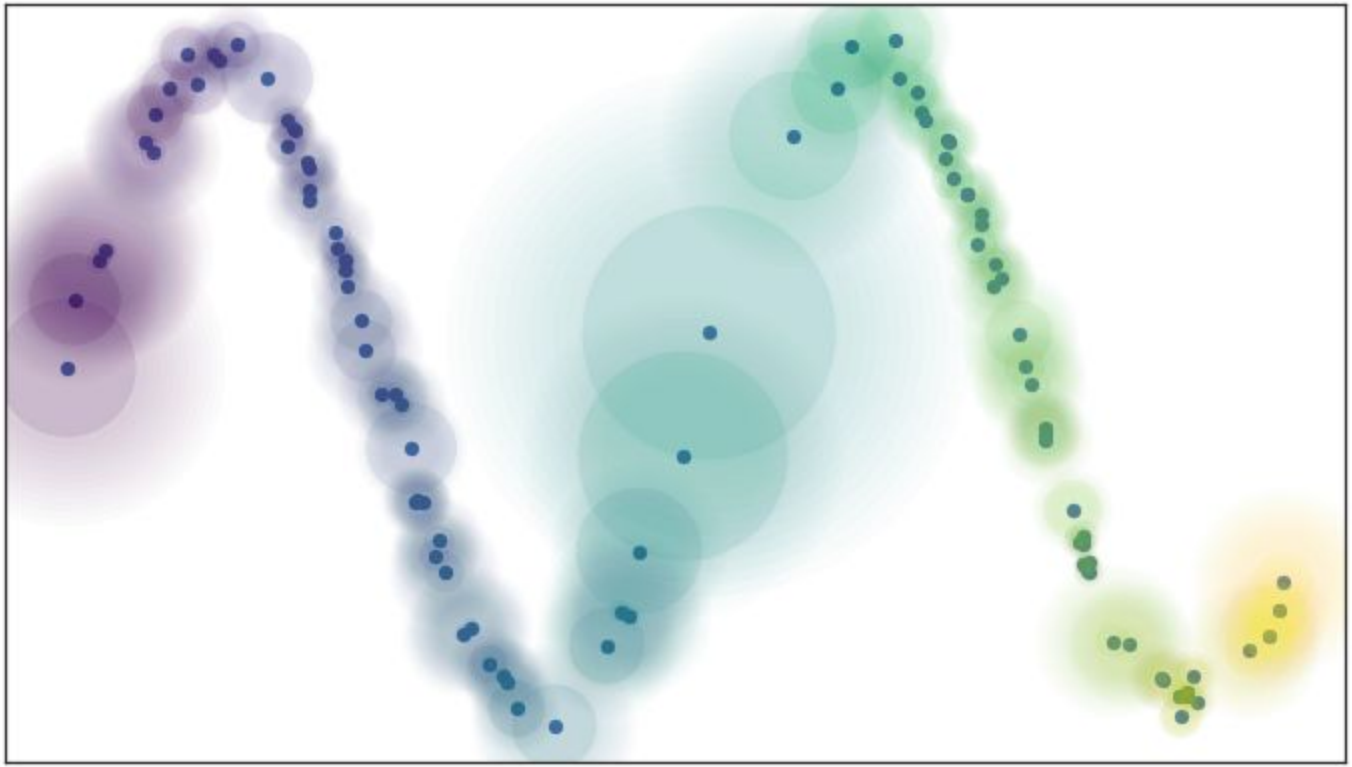
Open balls over uniformly distributed data



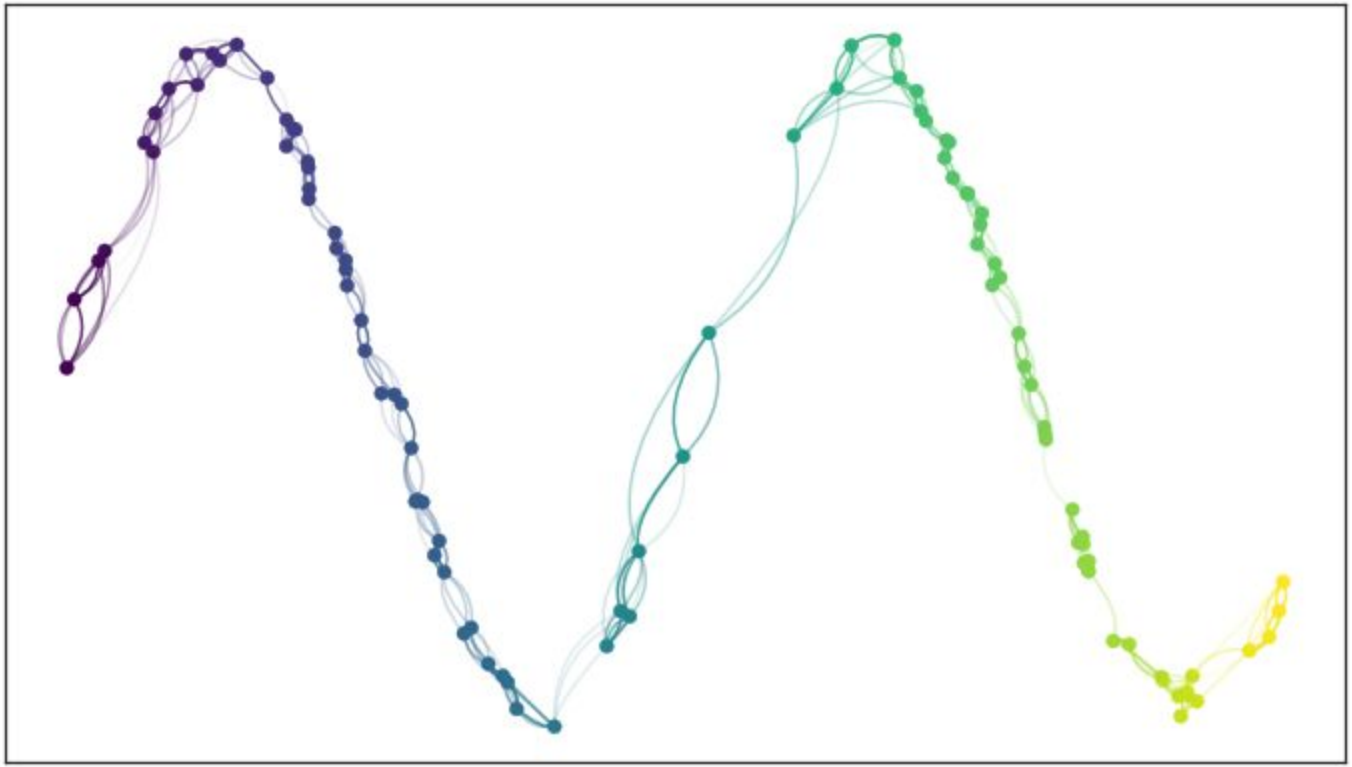
Open balls of radius one with a locally varying metric



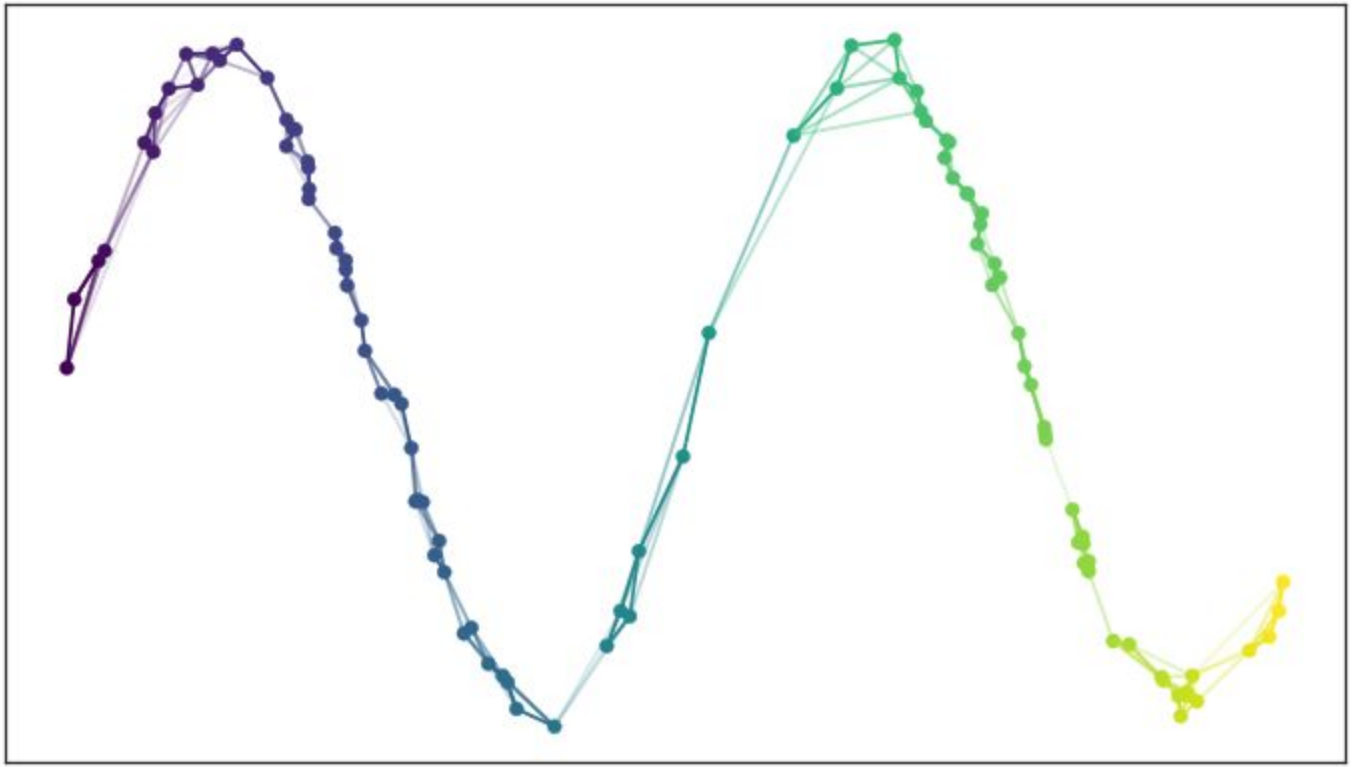
Fuzzy open balls of radius one with a locally varying metric



Local connectivity and fuzzy open sets



Edges with incompatible weights



Graph with combined edge weights

Having determined the weights we can optimise

- Colour on previous slides proportional to weight
- Move from a high (h) to a low (l) dimensional representation by minimizing the cross entropy
- In algorithm, set ‘min-dist’ and ‘n-neighbours’

Weight function

$$\sum_{e \in E} \underbrace{w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right)}_{\text{Attractive force}} + \underbrace{(1 - w_h(e)) \log\left(\frac{1 - w_h(e)}{1 - w_l(e)}\right)}_{\text{Repulsive force}}$$

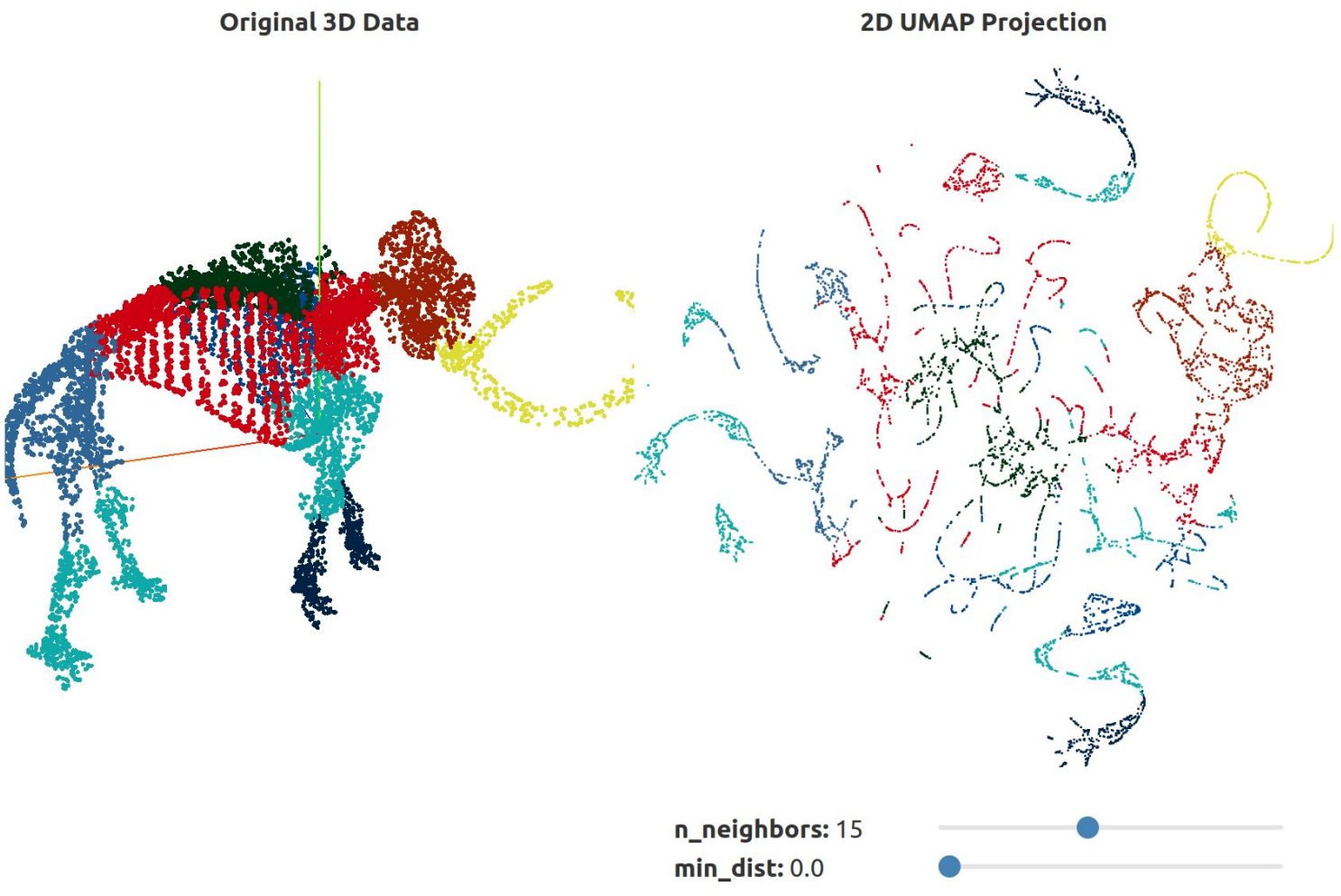


Figure 5: *UMAP projections of a 3D woolly mammoth skeleton (50k points, 10k shown) into 2 dimensions, with various settings for the* `n_neighbors` *and* `min_dist` *parameters.*

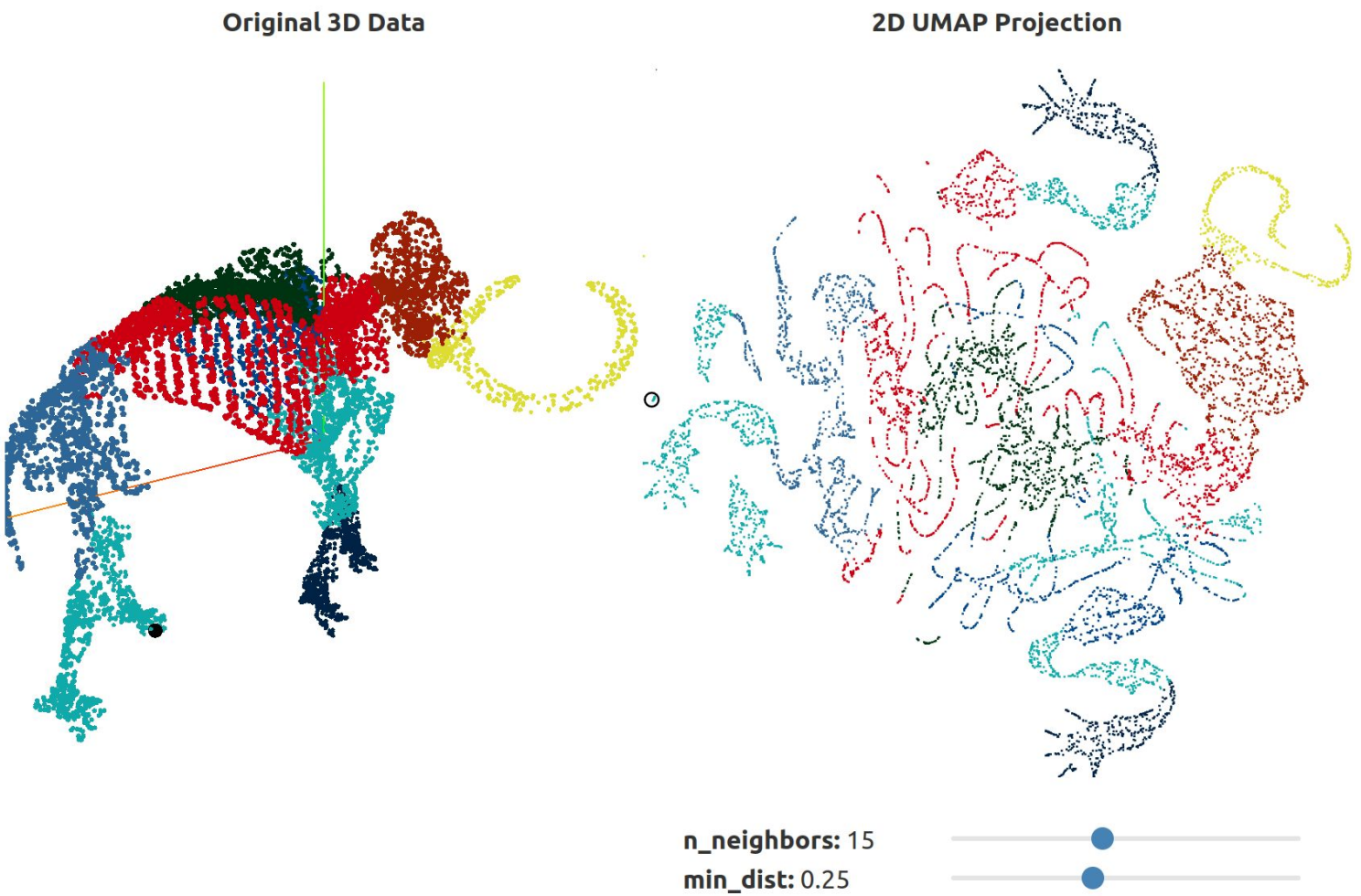
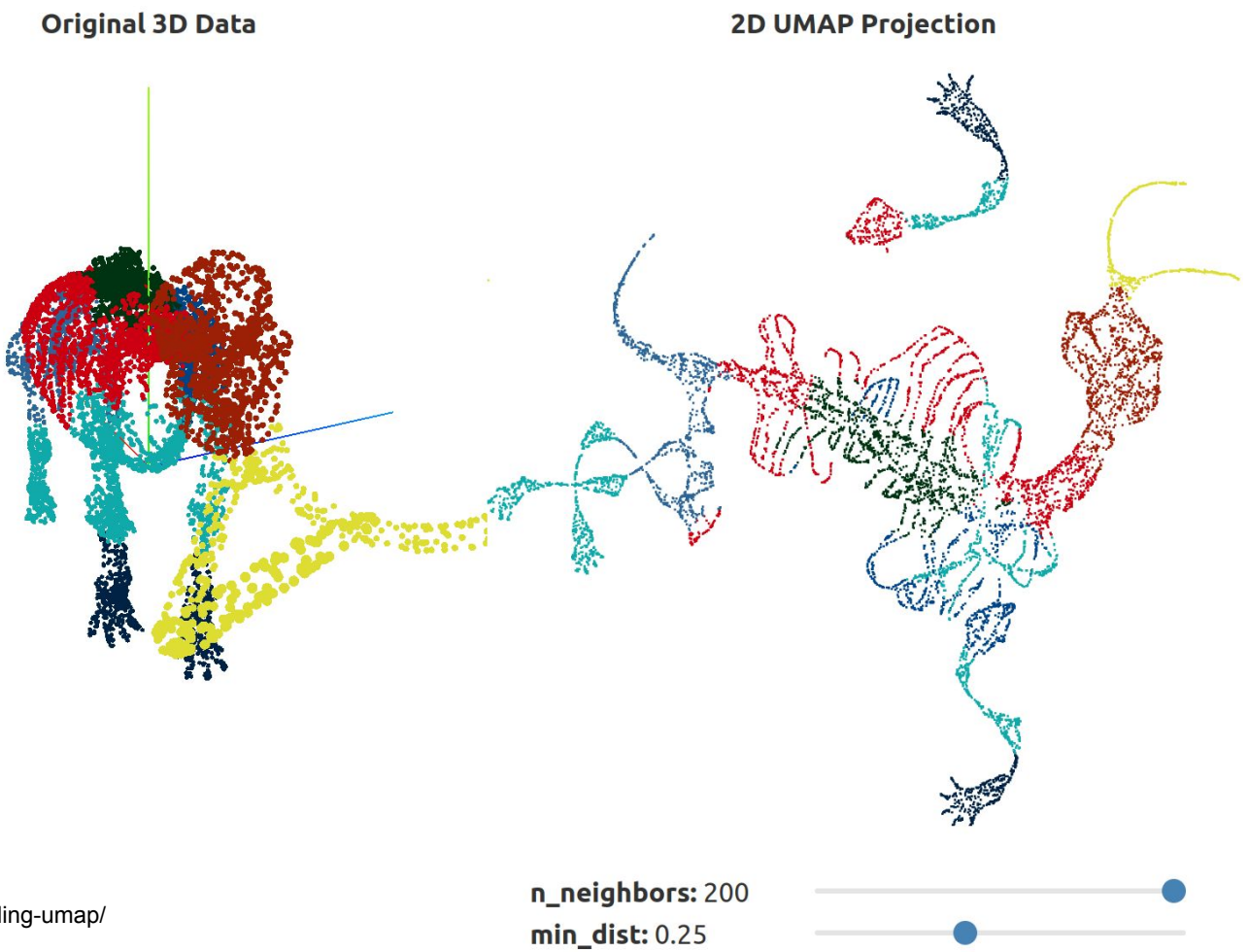


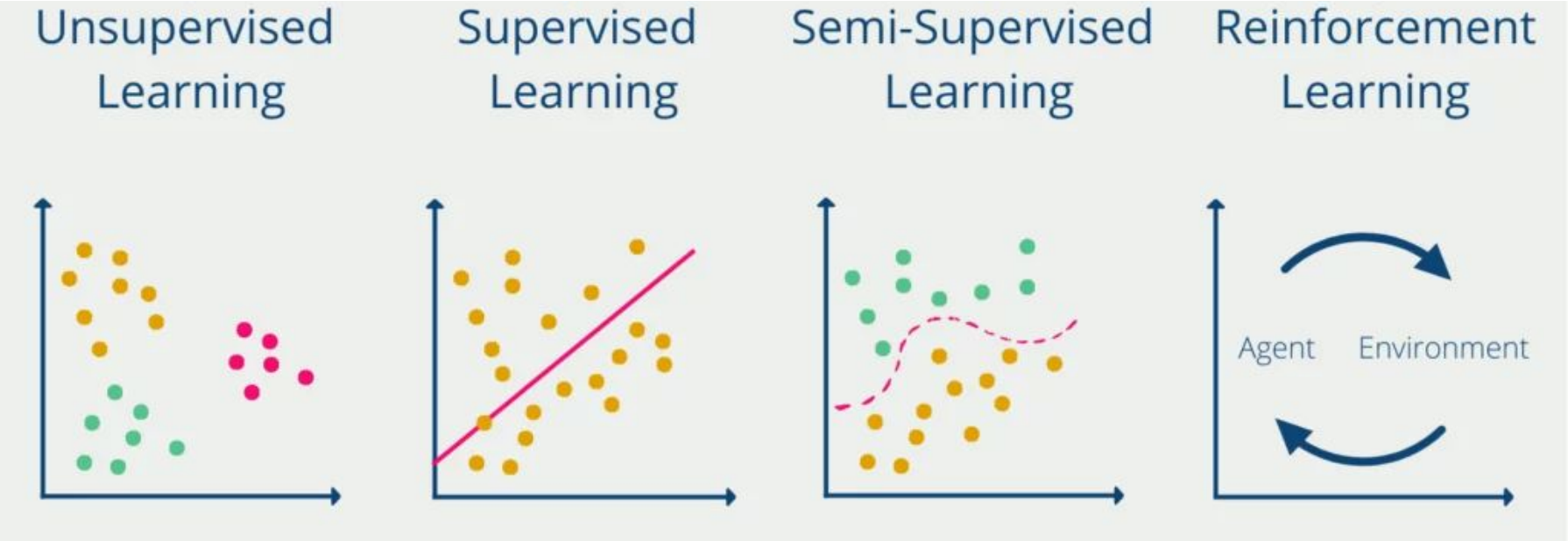
Figure 5: *UMAP projections of a 3D woolly mammoth skeleton (50k points, 10k shown) into 2 dimensions, with various settings for the* `n_neighbors` *and* `min_dist` *parameters.*



https://pair-code.github.io/understanding-umap/

Figure 5: *UMAP projections of a 3D woolly mammoth skeleton (50k points, 10k shown) into 2 dimensions, with various settings for the* `n_neighbors` *and* `min_dist` *parameters.*

Machine learning categories



Reinforcement learning

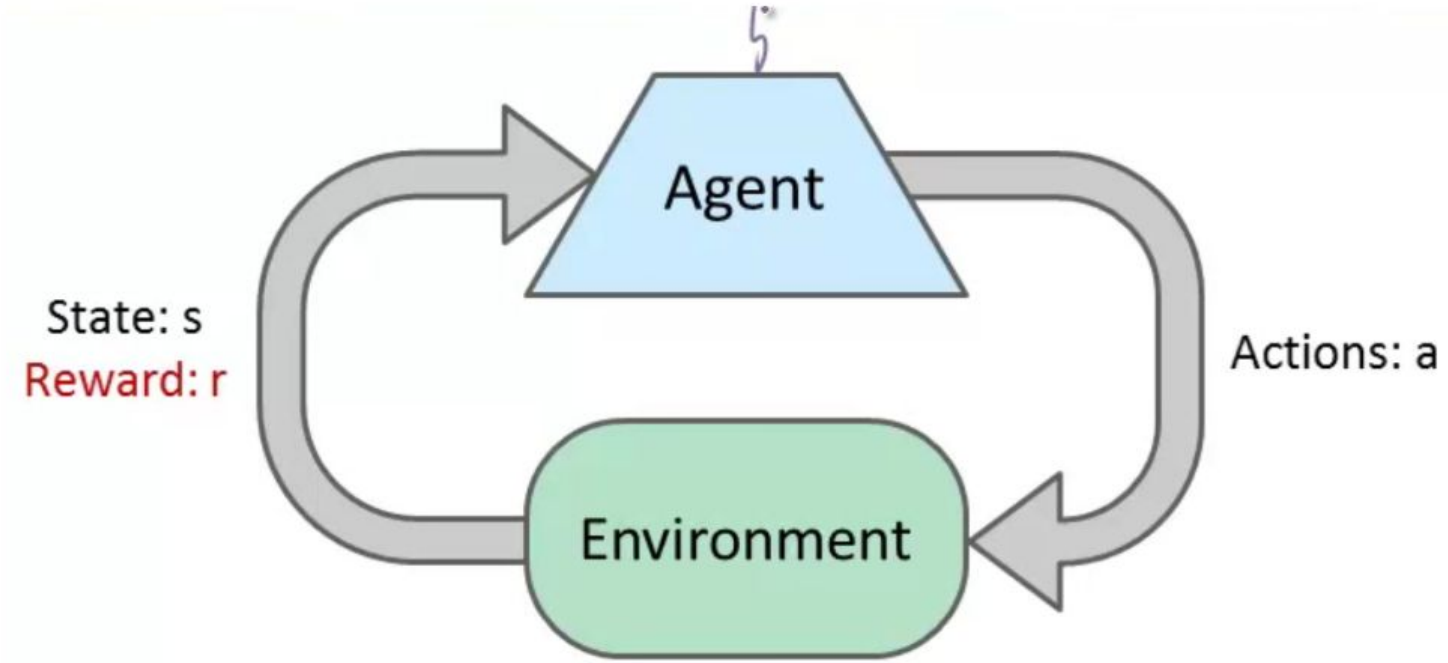
Learning algorithms differ in the information available to model we are developing:

- Supervised: correct outputs
- Unsupervised: no feedback, must construct measure of good output
- Reinforcement learning

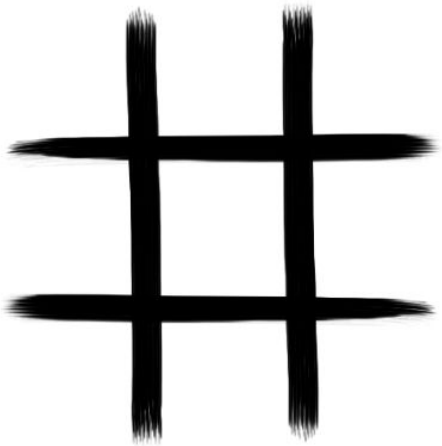
More realistic learning scenario:

- Continuous stream of input information, and actions
- Effects of action depend on state of the world
- Obtain reward that depends on world state and actions
- Not “correct” answers… just some feedback

Reinforcement learning

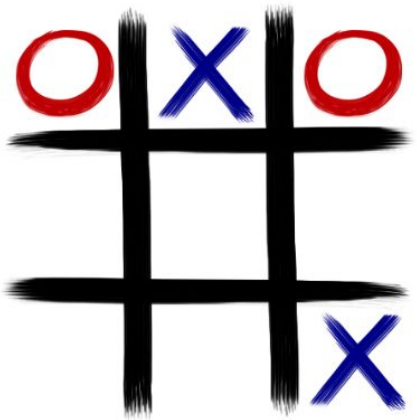


Reinforcement learning: Tic Tac Toe example, Notation

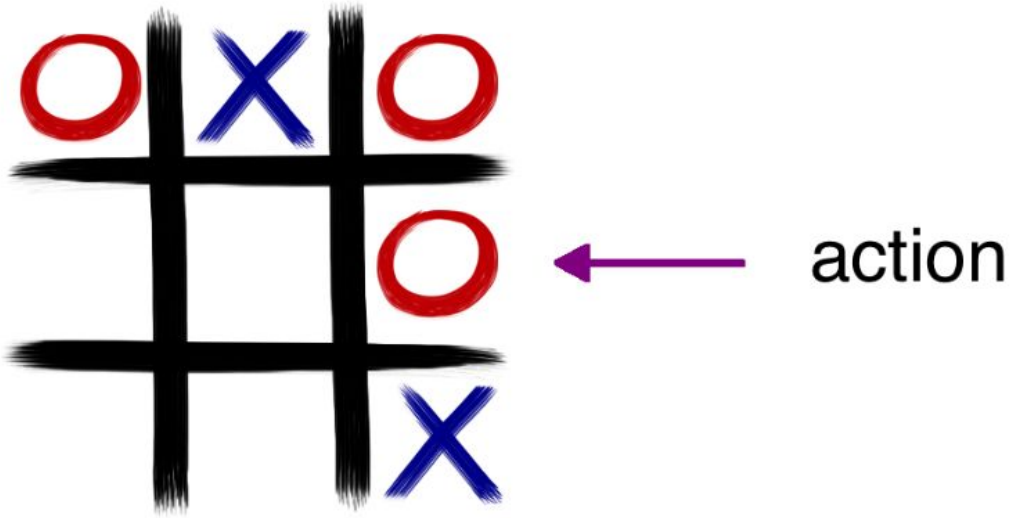


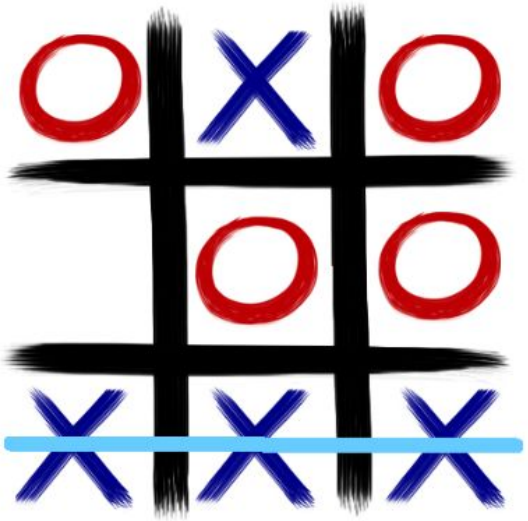
environment

Reinforcement learning: Tic Tac Toe example, Notation



(current)
state


action



reward

(here: -1)

Formulating Reinforcement Learning

- World described by a discrete, finite set of states and actions
- At every time step t, we are in a state s_t, and we:
  - Take an action at (possibly null action)
  - Receive some reward $\mathbf{r_{t+1}}$
  - Move into a new state $\mathbf{s_{t+1}}$
- An RL agent may include one or more of these components:
  - **Policy π**: agent's behaviour function
  - **Value function:** how good is each state and/or action
  - **Model:** agent's representation of the environment

Reinforcement Learning: Policy

- A policy is the agent's behaviour.
- It's a selection of which action to take, based on the current state
- Deterministic policy: $a = \pi(s)$
- Stochastic policy: $\pi(a|s) = P[a_t = a|s_t = s]$

Reinforcement Learning: Value Function

- Value function is a prediction of future reward
- Used to evaluate the goodness/badness of states
- Our aim will be to maximize the value function (the total reward we receive over time):
  - find the policy with the highest expected reward
- By following a policy π, the value function is defined as:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

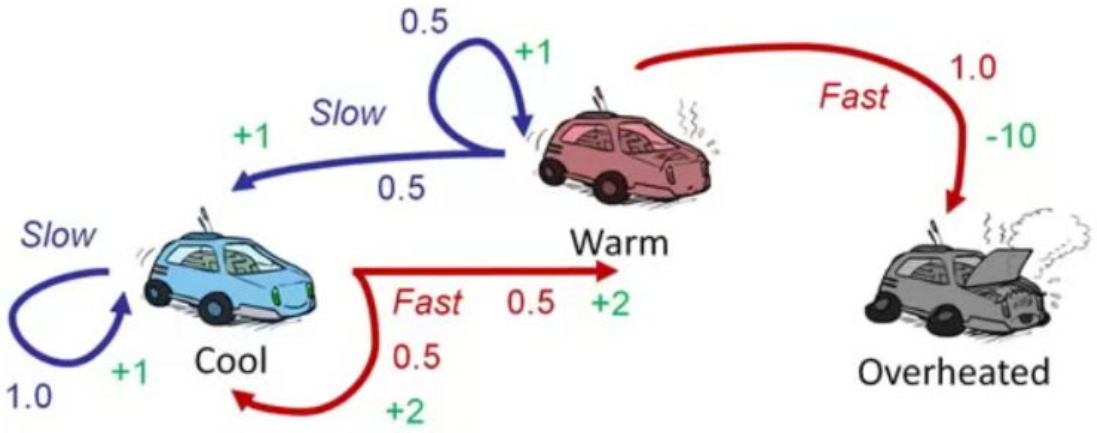
- γ is called a discount rate, and it is always $0 \leq \gamma \leq 1$
- If γ is close to 1, rewards further in the future count more, and we say that the agent is "farsighted"
- If γ is less than 1 because there is usually a time limit to the sequence of actions needed to solve a task (we prefer rewards sooner rather than later)

Reinforcement Learning: Model

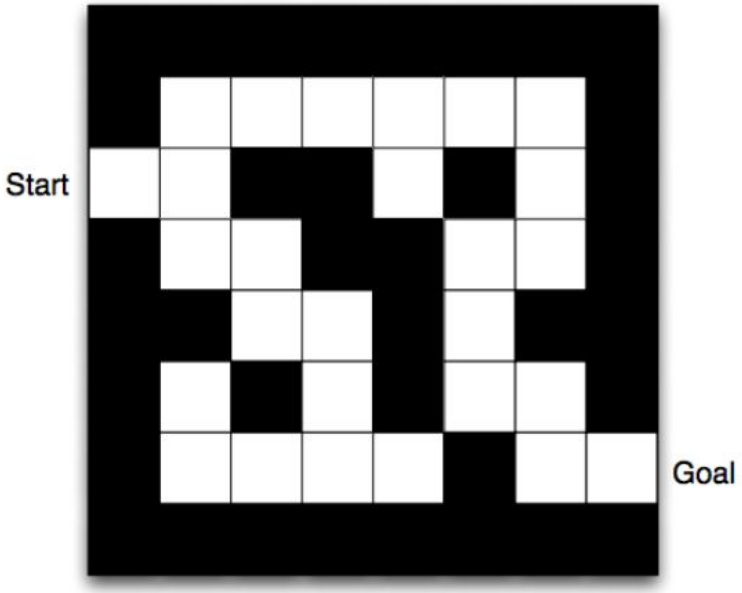
The model describes the environment by a distribution over rewards and state transitions:

$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

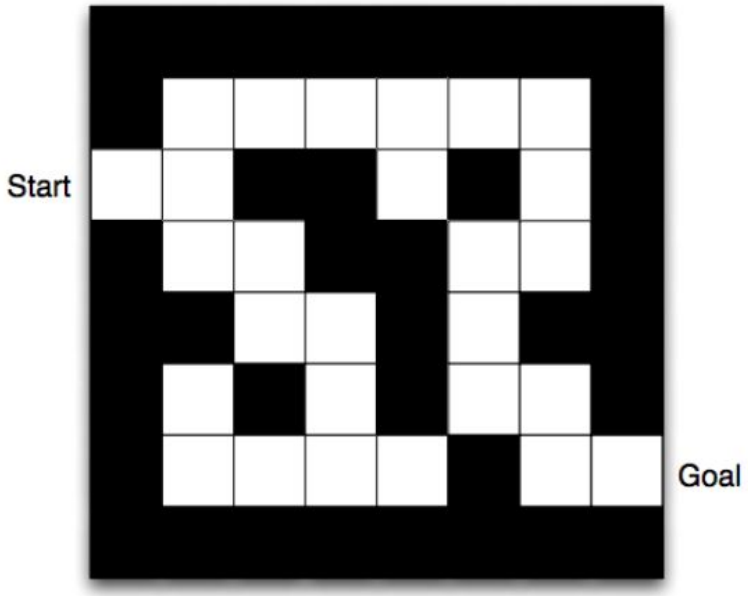
We assume the Markov property: the future depends on the past only through the current state



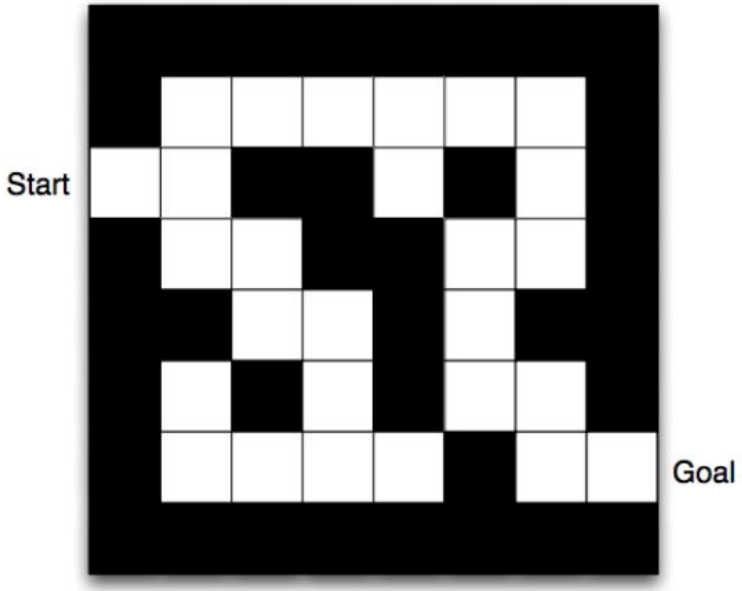
M



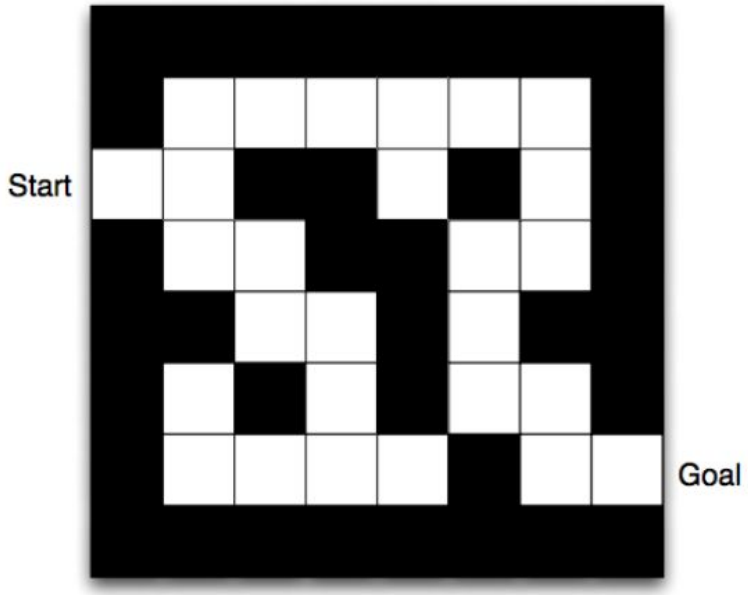
- Rewards:



- Rewards: -1 per time-step
- Actions:

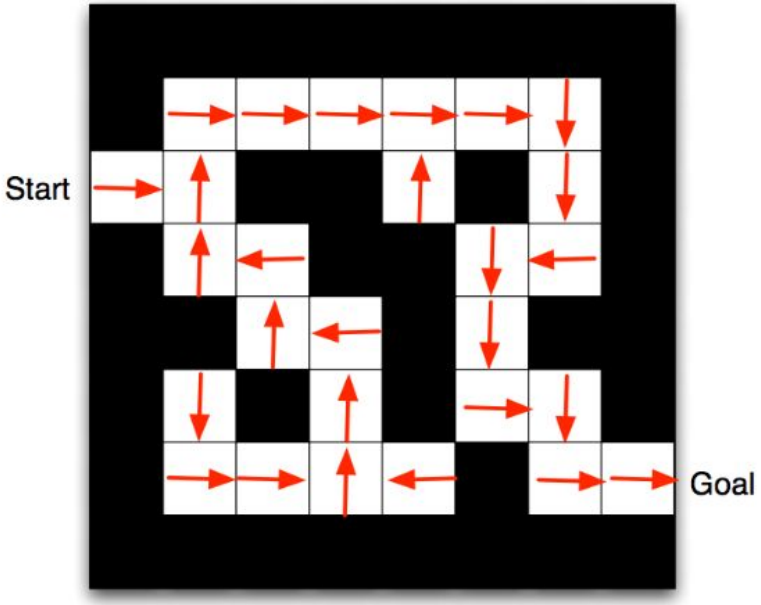


- Rewards: -1 per time-step
- Actions: N, E, S, W
- States:



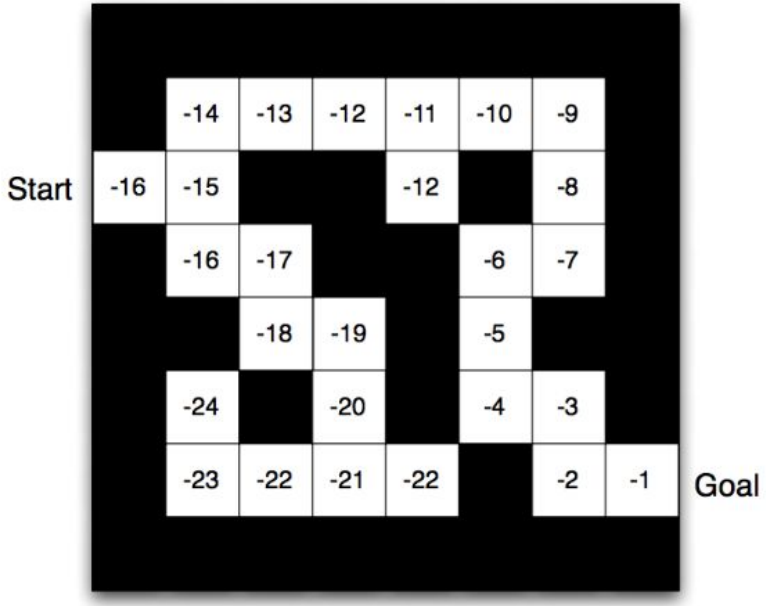
- Rewards: -1 per time-step
- Actions: N, E, S, W
- States: Agent's location

[Slide credit: D. Silver]



- Arrows represent policy $\pi(s)$ for each state s

[Slide credit: D. Silver]



- Numbers represent value $V^{\pi}(s)$ of each state s

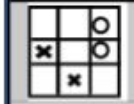
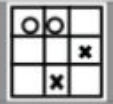
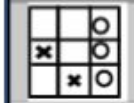
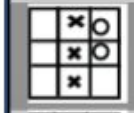
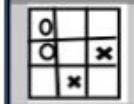
[Slide credit: D. Silver]

Example: Tic-Tac-Toe

- Consider the game tic-tac-toe:
  - Reward: win/lose/tie the game (+1/ − 1/0) [only at final move in given game]
  - State: positions of X’s and O’s on the board
  - Policy: mapping from states to actions
    - Based on rules of game: choice of one open position
  - Value function: prediction of reward in future, based on current state
- In tic-tac-toe, since state space is tractable, can use a table to represent value function

Example: Tic-Tac-Toe

- Each board position (taking into account symmetry) has some probability
- Simple learning process:
  - start with all values = 0.5
  - policy: choose move with highest probability of winning given current legal moves from current state
  - update entries in table based on outcome of each game
  - After many games value function will represent true probability of winning from each state
- Can try alternative policy: sometimes select moves randomly (exploration!)

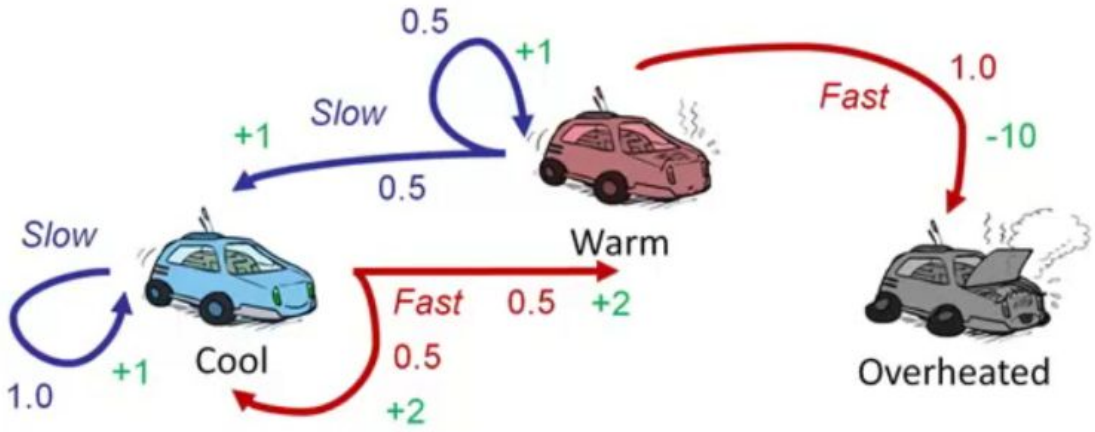
State	Probability of a win (Computer plays "o")
	0.5
	0.5
	1.0
	0.0
	0.5
etc	

Basic Problems

- Markov Decision Problem (MDP): tuple (S, A, P, γ) where P is

$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

- Standard MDP problems:
  - 1. Planning: given complete Markov decision problem as input, compute policy with optimal expected return



Basic Problems

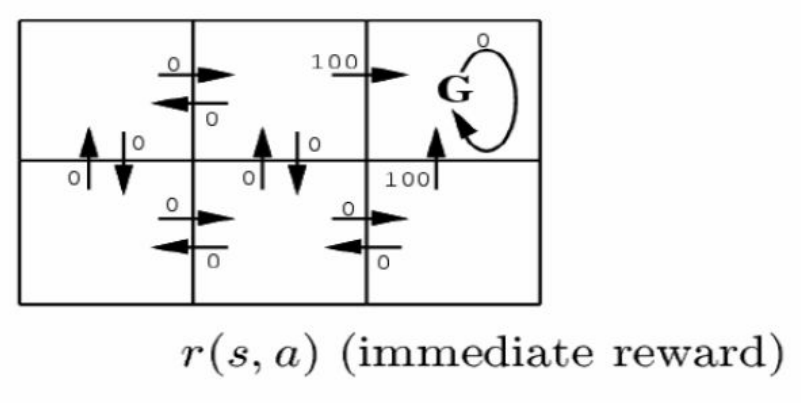
- Markov Decision Problem (MDP): tuple (S, A, P, γ) where P is

$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

- Standard MDP problems:
  - 1. Planning: given complete Markov decision problem as input, compute policy with optimal expected return
  - 2. Learning: We don’t know which states are good or what the actions do.
    - We must try out the actions and states to learn what to do…

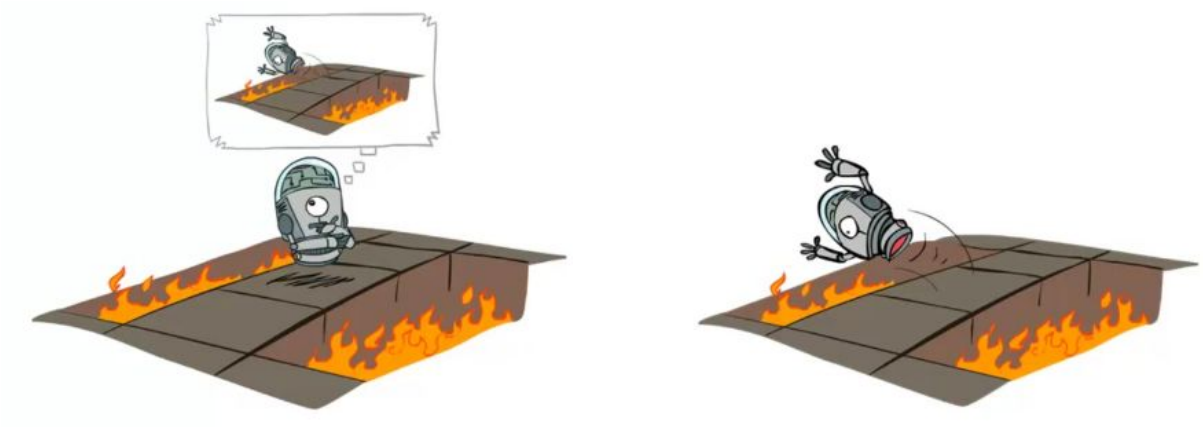
Example of Standard MDP Problem

- 1. Planning: given complete Markov decision problem as input, compute policy with optimal expected return
- 2. Learning: We don't know which states are good or what the actions do.
  - We must try out the actions and states to learn what to do…



Example of Standard MDP Problem

- 1. Planning: given complete Markov decision problem as input, compute policy with optimal expected return
- 2. Learning: We don’t know which states are good or what the actions do.
  - We must try out the actions and states to learn what to do…



We will focus on learning, but discuss planning along the way…

Exploration vs. Exploitation

- If we knew how the world works (embodied in P), then the policy should be deterministic
  - Select optimal action in each state
- Reinforcement learning is like trial-and-error learning
- The agent should discover a good policy from its experiences of the environment
- Without losing too much reward along the way
- Since we do not have complete knowledge of the world, taking what appears to be the optimal action may prevent us from finding better states/actions
- Interesting trade-off:
  - Immediate reward (exploitation) vs. gaining knowledge that might enable higher future reward (exploration)

Exploitation/Exploration examples

- Restaurant Selection
  - Exploitation: Go to your favourite restaurant
  - Exploration: Try a new restaurant
- Online Banner Advertisements
  - Exploitation: Show the most successful advert
  - Exploration: Show a different advert
- Oil Drilling
  - Exploitation: Drill at the best known location
  - Exploration: Drill at a new location
- Game Playing
  - Exploitation: Play the move you believe is best
  - Exploration: Play an experimental move

Markov Decision Problem (MDP) formulation

- Goal: find policy π that maximizes expected accumulated future rewards $V^{\pi}(s_t)$, obtained by following π from state s_t:

$$\begin{aligned} V^{\pi}(s_t) &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \\ &= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned}$$

- Game show example:
  - Assume series of questions, increasingly difficult, but increasing payoff
  - Choice: accept accumulated earnings and quit; or continue and risk losing everything
- Notice that:

$$V^{\pi}(s_t) = r_t + \gamma V^{\pi}(s_{t+1})$$

What to Learn?

- We might try to learn the function V (which we write as V*)

$$V^*(s) = \max_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

- Here δ(s, a) gives the next state, if we perform action a in current state s
- We could then do a lookahead search to choose best action from any state s:

$$\pi^*(s) = arg\ \max_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

- But there's a problem:
  - This works well if we know δ() and r ()
  - But when we don't, we cannot choose actions this way

Q Learning

- Define a new function very similar to V*

$$Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a))$$

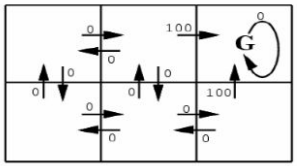
- If we learn Q, we can choose the optimal action even without knowing δ

$$\begin{aligned} \pi^*(s) &= arg\max_a [r(s,a) + \gamma V^*(\delta(s,a))] \\ &= arg\max_a Q(s,a) \end{aligned}$$

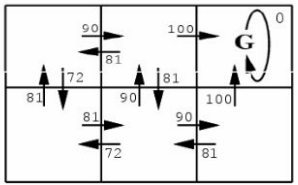
- Q is then the evaluation function we will learn

Q Learning

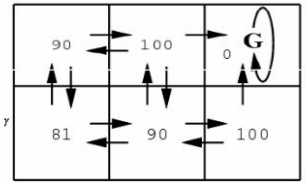
$\gamma = 0.9$



$r(s, a)$ (immediate reward) values

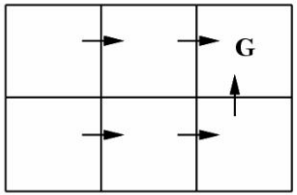


$Q(s, a)$ values



$V^*(s)$ values

$V^*(s_5) = 0 + \gamma 100 + \gamma^2 0 + \ldots = 90$



One optimal policy

Training Rule to Learn Q

- Q and V* are closely related:

$$V^*(s) = \max_a Q(s, a)$$

- So we can write Q recursively:

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t)) \\ &= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

- Let $\hat{Q}$ denote the learner's current approximation to Q
- Consider training rule, where s′ is state resulting from applying action a in state s

$$\hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')$$

Q Learning for Deterministic World

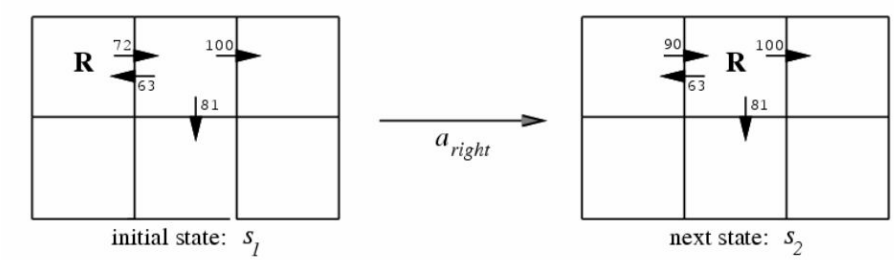
- For each s, a initialize table entry $\hat{Q}$(s, a) $\leftarrow$ 0
- Start in some initial state s
- Do forever:
  - Select an action a and execute it
  - Receive immediate reward r
  - Observe the new state s′
  - Update the table entry for $\hat{Q}$(s, a) using Q learning rule:

$$\hat{Q}(s,a) \leftarrow r(s,a) + \gamma \max_{a'} \hat{Q}(s',a')$$

  - s $\leftarrow$ s′
- If we get to absorbing state, restart to initial state, and run thru “Do forever” loop until reach absorbing state

Updating Estimated Q

- Assume the robot is in state s1; some of its current estimates of Q are as shown; executes rightward move



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$
$$\leftarrow r + 0.9 \max_{a}\{63, 81, 100\} \leftarrow 90$$

- Important observation: at each time step (making an action a in state s only one entry of will $\hat{Q}$ change (the entry $\hat{Q}$(s, a))
- Notice that if rewards are non-negative, then $\hat{Q}$ values only increase from 0, approach true Q

Q Learning: Summary

- Training set consists of series of intervals (episodes): sequence of (state, action, reward) triples, end at absorbing state
- Each executed action a results in transition from state s_i to s_j; algorithm updates $\hat{Q}(s_i, a)$ using the learning rule
- Intuition for simple grid world, reward only upon entering goal state → Q estimates improve from goal state back
  - 1. All $\hat{Q}(s, a)$ start at 0
  - 2. First episode – only update $\hat{Q}(s, a)$ for transition leading to goal state
  - 3. Next episode – if go thru this next-to-last transition, will update $\hat{Q}(s, a)$ another step back
  - 4. Eventually propagate information from transitions with non-zero reward throughout state-action space

Q Learning: Exploration/Exploitation

- Have not specified how actions chosen (during learning)
- Can choose actions to maximize $\hat{Q}$(s, a)
- Good idea?
- Can instead employ stochastic action selection (policy):

$$P(a_i|s) = \frac{\exp(k\hat{Q}(s, a_i))}{\sum_j \exp(k\hat{Q}(s, a_j))}$$

- Can vary k during learning
  - More exploration early on, shift towards exploitation