Advanced Dimensionality Reduction Techniques
CS 446/546

Outline

- Overview / PCA recap
- SOM (self-organizing maps)
- AE (autoencoder)
- Spectral Clustering
- ISOMAP

Introduction

- Most traditional statistical techniques (e.g. regression/classification) were developed in *low-dimensional* settings (i.e. $n >> p$ where n is the data size and p is the number of features).

- Over the last several decades, new technologies have drastically changed the way that data are collected (see "big data age"). Consequently, it is now commonplace to work with data with a very large number of features (i.e. $p >> n$).

- While p can be extremely large, the number of observations n is often limited due to cost, sample availability, or other considerations.

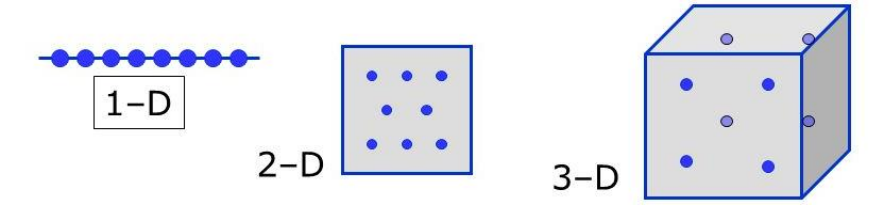
Introduction

- Data containing more features than observations are typically referred to as ***high-dimensional***.

- Issues pertaining to the *bias-variance tradeoff* and *overfitting* are commonly exacerbated in high dimensions.

- With a large number of features, statistical models (e.g. regression) can become too flexible and hence overfit the data.

- Recall the *curse of dimensionality*, which poses two fundamental, associated problems: (1) "neighborhoods" become very large (this is problematic in particular for kernel and clustering methods), (2) we need a much larger data set to adequately "fill" the space for predictive modeling, etc.



Interpretability in High Dimensions

- In high-dimensional settings we need to be **cautious about how we interpret our results** – that is to say if they can be reasonably interpreted at all.

- Of course, it is oftentimes adequate, depending on the application, to treat a machine learning model as a *mere* predictive "black box" (e.g. statistical arbitrage, government work).

- Conversely, if we want to say that the features in our model directly impact the outcomes we observe (note: in ML we almost never use the c-word – viz., variables *caused* observed effect) we need to be alert to ***multicollinearity***.

- In high dimensions, it is very likely that some of our model variables are mutually correlated. This means we can never know exactly which variables (if any) are truly predictive of the outcome. Moreover, we can rarely identify the optimal set of features for a given phenomenon of interest.

Interpretability in High Dimensions

- The "first rule" of data science and ML: **one can always add more and more features to achieve zero classification/predictive error**, a perfect correlation coefficient value, etc.

- In the end, however, ***this is a useless model***. We always need to report results on an independent test or validation set.

- In 2008, Hinton *et al*, developed a non-linear dimensionality technique known as ***t-SNE*** (*t-distributed stochastic neighbor embedding*) that is particularly well-suited for embedding high-dimensional data into 2 or 3 dimensions, which can be visualized with a scatter plot.

- Specifically, it models each high-dimensional object by a two- or three-dimensional point in such a way that **similar objects are modeled by nearby points** and dissimilar objects are modeled by distant points.

Dimensionality Reduction

- In general: the higher the number of dimensions we have, the more training data we need.
- Additionally, computational cost is generally an explicit function of dimensionality.
- Dimensionality reduction can also **remove noise** in a data set, which can, in turn, significantly improve the results of a learning algorithm.
- These are perhaps the strongest reasons why dimensionality reduction is useful (in addition to improving visualization/interpretability).

In general, there are (3) common ways to perform dimensionality reduction:

(1) Feature selection – determine whether the features available are actually useful, i.e. are they correlated with the output variables.

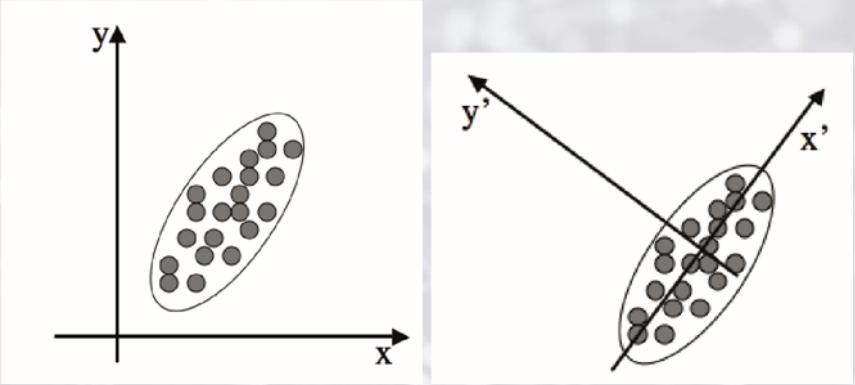
(2) Feature derivation – means deriving new features from old ones, generally by applying transforms to the data set that change the coordinate system axes (e.g., by moving or rotating); this is usually achieved through matrix multiplication.

(3) Clustering – group together similar data points to see whether this allows fewer features to be used.

PCA

PCA generates a particular set of coordinate axes **that capture the maximum variability in the data**; furthermore, these new coordinate axes are **orthogonal.**

The figure shows two versions of the same data set.



- In the **first image**, the data are arranged in an ellipse that runs at 45° axes; while in **the second**, the axes have been moved so that the data now runs along the x-axis and is centered on the origin.

- Key idea: the potential for dimensionality reduction rests in the fact that the *y* dimension now does not demonstrate much variability – and so it might be possible to ignore it and simply use the *x* axis values alone for learning, etc.

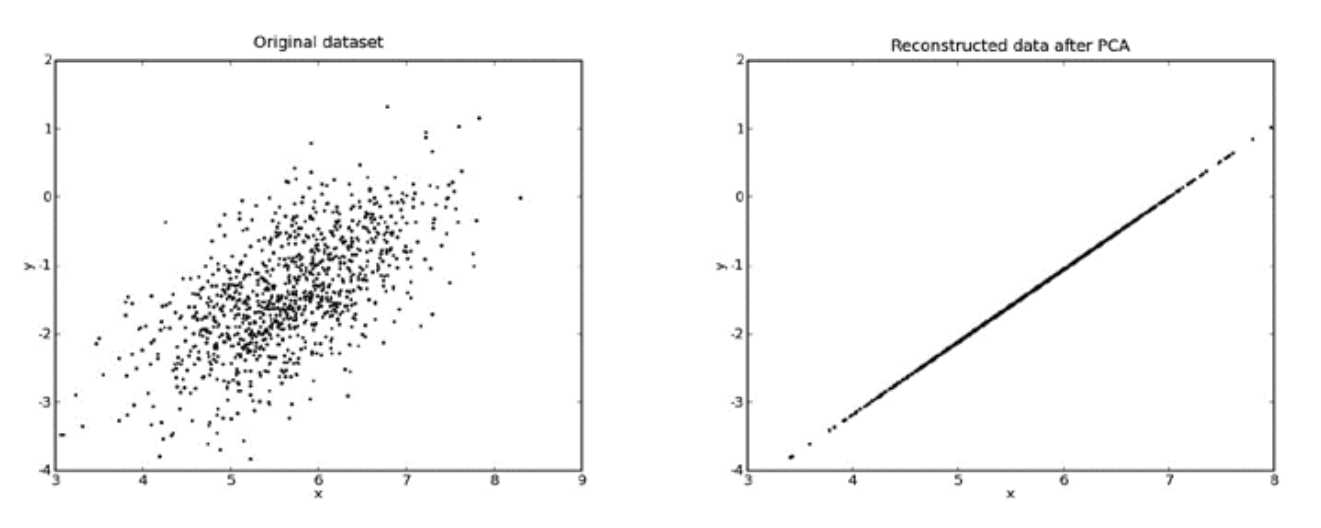
(*) In fact, applying this dimensionality reduction often has the nice effect of removing some of the noise in the data.

PCA

$$\mathbf{Z} = \mathrm{cov}(\mathbf{X}) = \mathbf{EDE}^T$$

- Note: In the eigendecomposition for cov(**X**), the dimensions with large eigenvalues have lots of variation and are therefore useful dimensions.

- In order to perform a **dimensionality reduction** on our data set, we can therefore throw away dimensions for which the eigenvalues are very small (usually smaller than some chosen parameter).



PCA

- Here is the **PCA algorithm**:

(1) Write N data points $x_i=(x_{1i},x_{2i},\ldots,x_{Mi})$ as row vectors.

(2) Put these vectors into the data matrix **X** (of size N x M).

(3) Center the data by subtracting off the mean of each column, place into matrix **B**.

(4) Computer the covariance matrix: $\mathbf{C} = \frac{1}{N}\mathbf{B}\mathbf{B}^T$

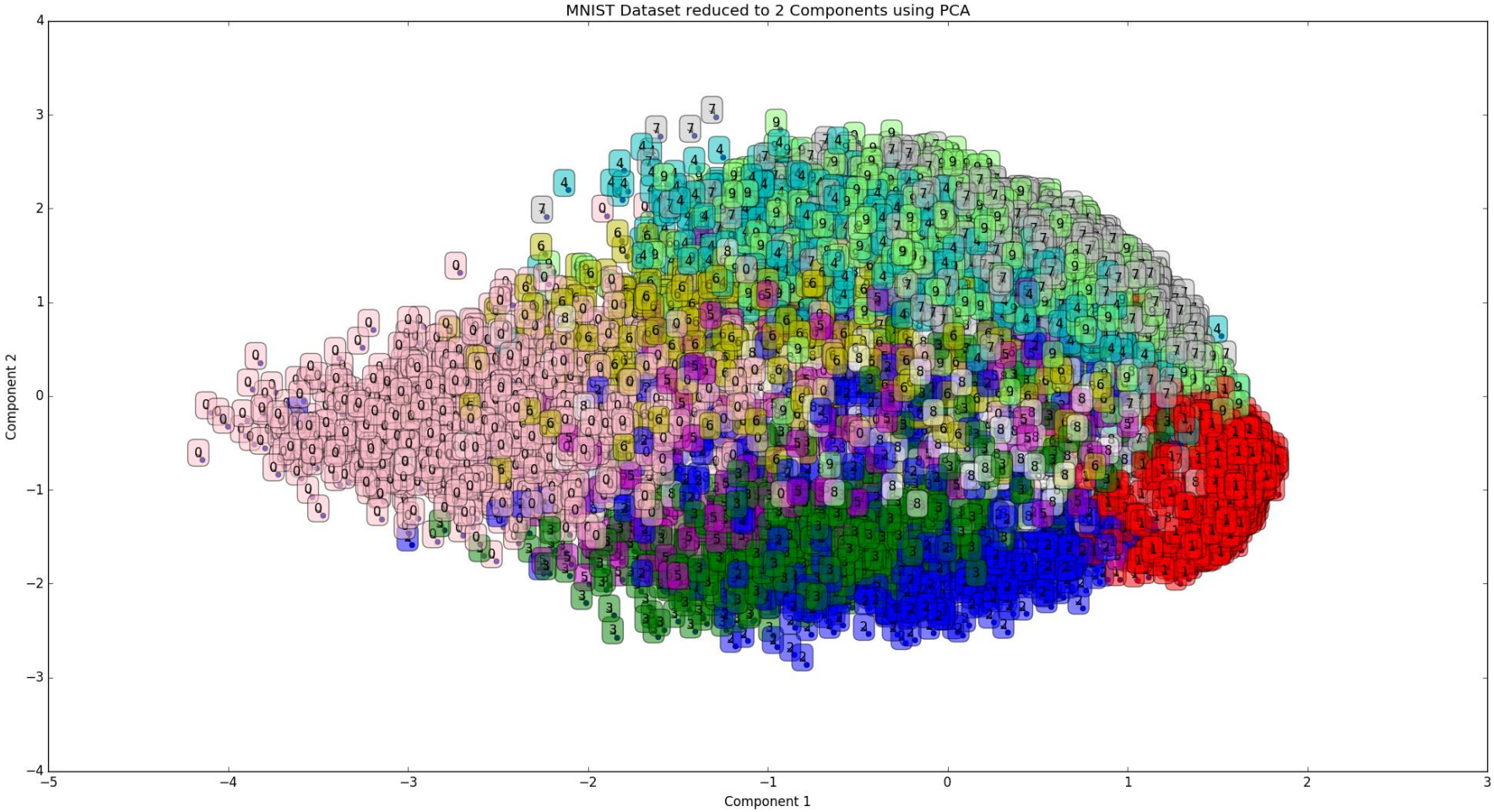
(5) Computer the *eigenvalues* and *eigenvectors* of **C**, so: $\mathbf{C} = \mathbf{V}\mathbf{D}\mathbf{V}^T$

where **D** is the diagonal matrix of eigenvalues; V is the matrix of corresponding eigenvectors.

(6) Sort of the columns of D into order of decreasing eigenvalues, and apply the same order to the columns of V.

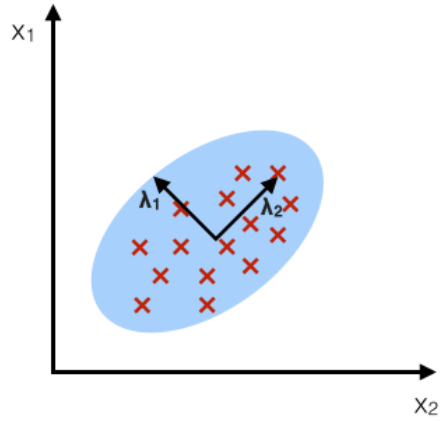
(7) Reject those with eigenvalues less than some given threshold, leaving L dimensions in the data.

PCA for MNIST

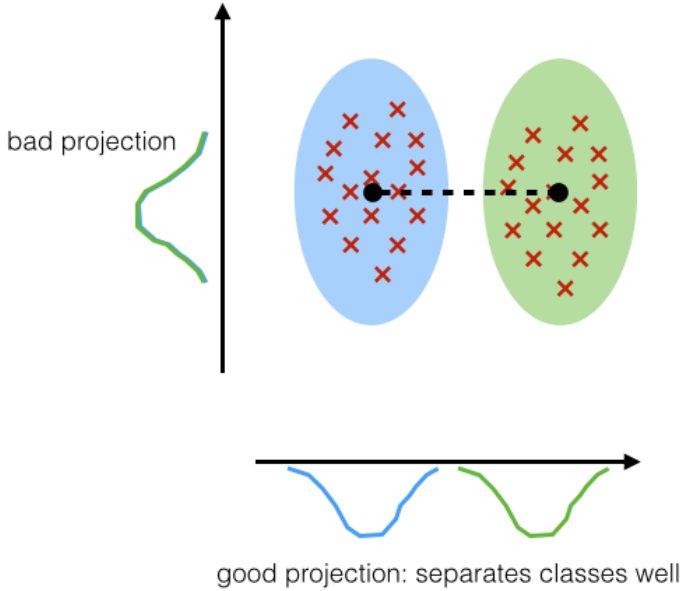


PCA vs. LDA

PCA:
component axes that maximize the variance



LDA:
maximizing the component axes for class-separation



Extending PCA

Q: What strong assumptions did we make about the surface for the directions of maximum variation with PCA?

A: We assumed these surfaces of maximum variation are **straight lines** (this is a strong assumption!)

Q: How can are break the linear restriction for PCA?

A: **"Kernelize" PCA**!

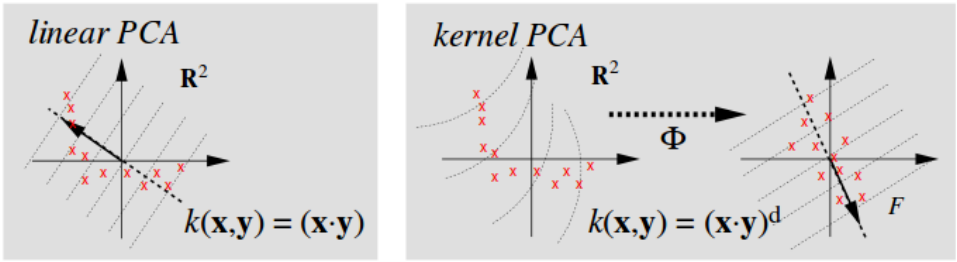


Fig. 1. Basic idea of kernel PCA: by using a nonlinear kernel function k instead of the standard dot product, we implicitly perform PCA in a possibly high–dimensional space F which is nonlinearly related to input space. The dotted lines are contour lines of constant feature value.

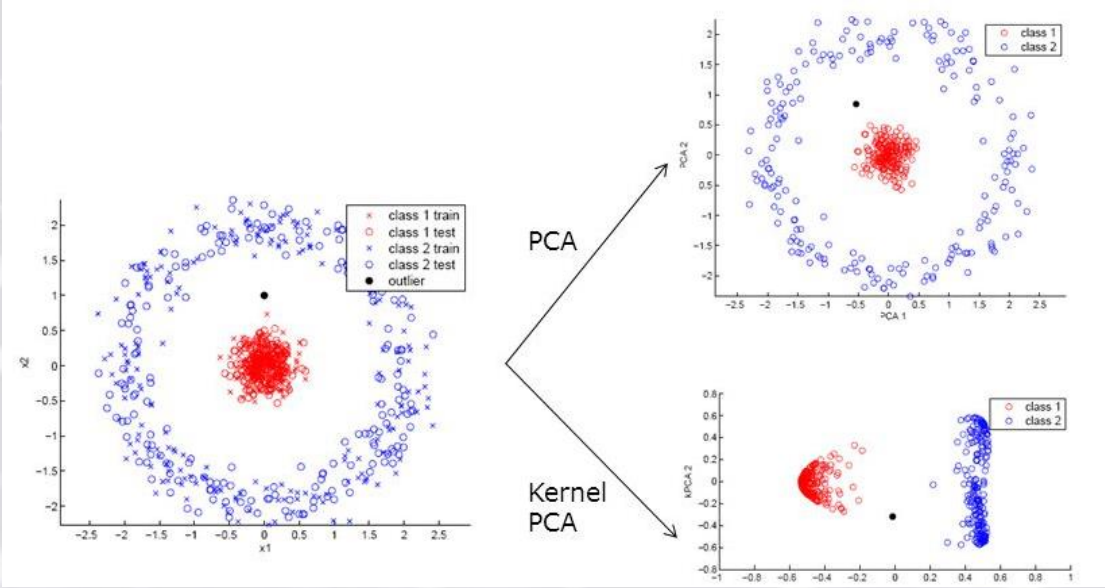
Kernel PCA

(*) All we have to do is express the covariance matrix **C** (recall this was the covariance of the data matrix X after centering) in terms of a kernel transformation:

$$\mathbf{C} = \frac{1}{N}\sum_{i=1}^{N} \Phi(x_n) \bullet \Phi(x_n)^T$$

(*) Next we compute the *eigendecomposition* of **C** and use the eigenvectors with the largest associated eigenvalues for PCA.

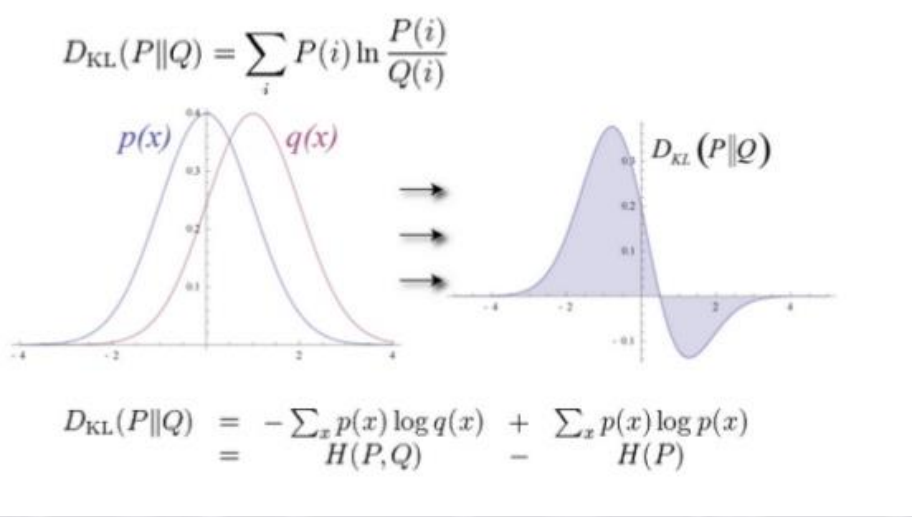
(*) Recall (from SVM lecture) that by using a kernel function we implicitly perform a dot product in a larger dimensional feature space (this is the crux of the **kernel trick**), with the upshot of enhanced expressiveness.



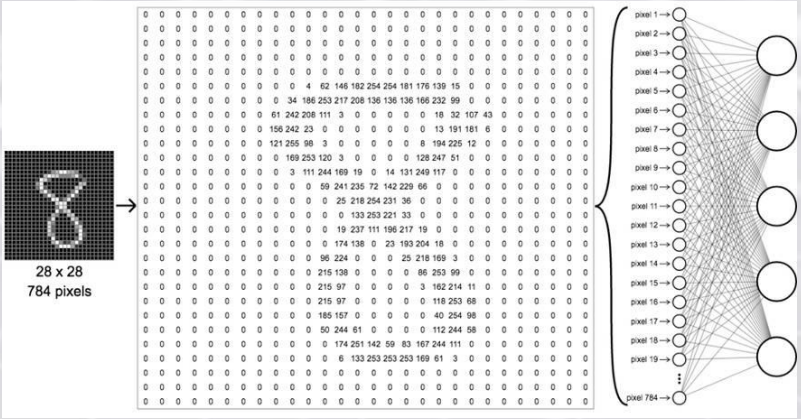
t-SNE: H-D Data Visualization

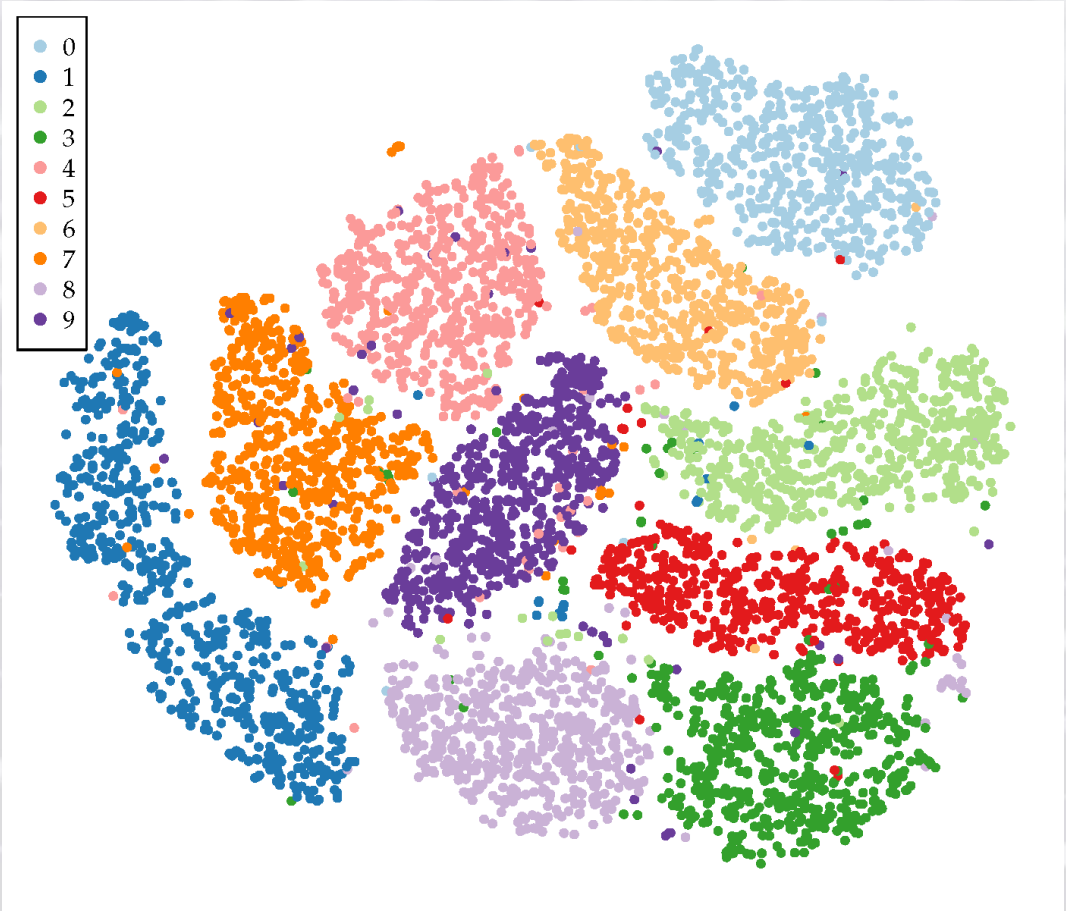
- First, t-SNE constructs a probability distribution over pairs of high-dimensional objects in such a way that similar objects have a high probability of being picked, whilst dissimilar points have an extremely small probability of being picked.

- Second, t-SNE defines a similar probability distribution over the points in the low-dimensional map, and it minimizes the ***KL divergence*** (a standard measure of "distance" between probability distributions) between the two distributions with respect to the locations of the points in the map.

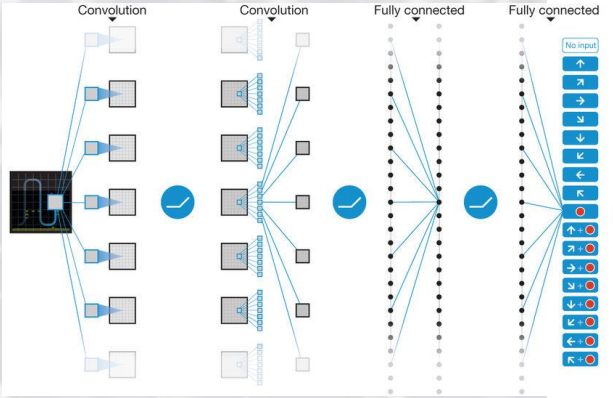


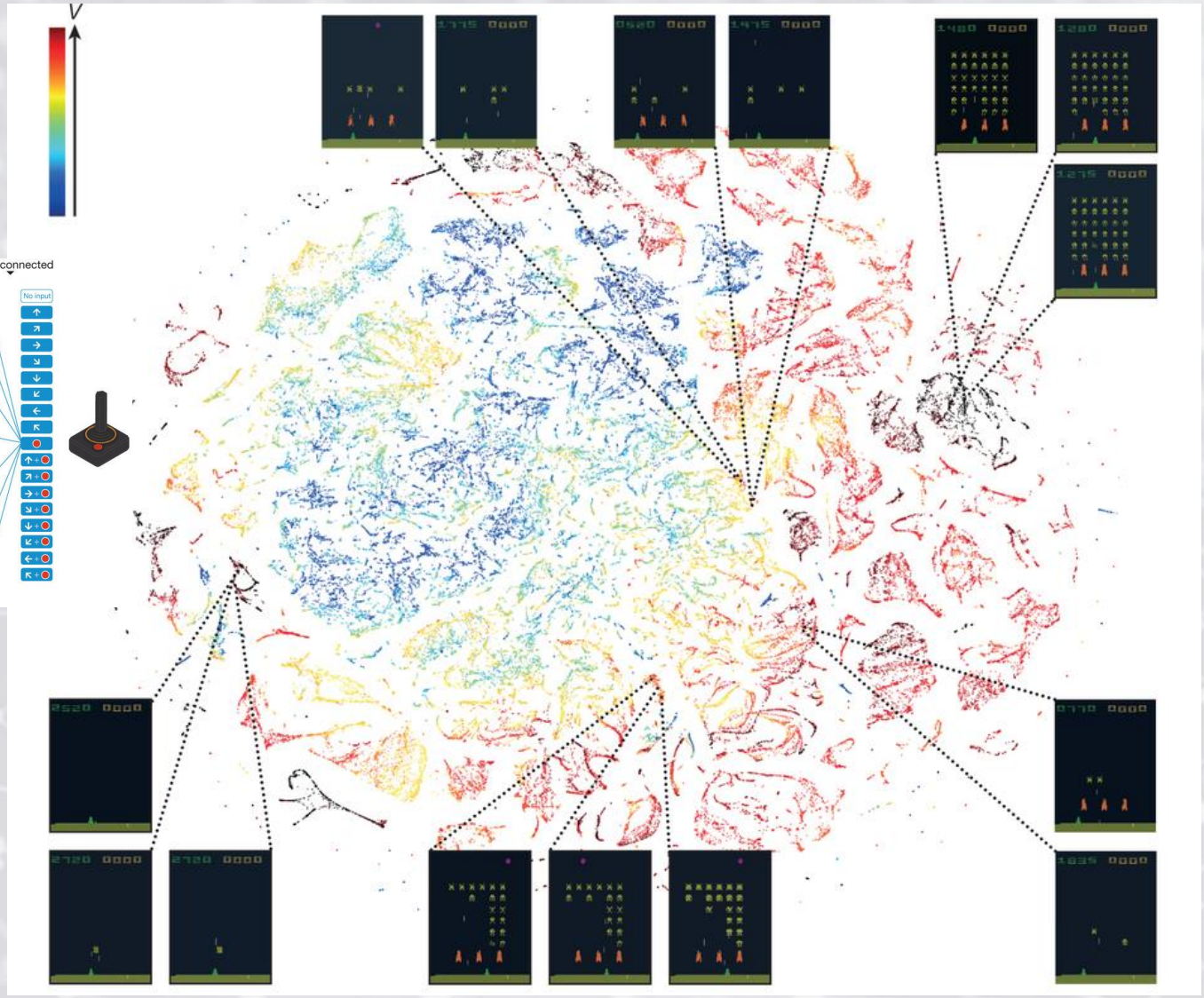
t-SNE for MNIST





t-SNE for Atari! (Deepmind)





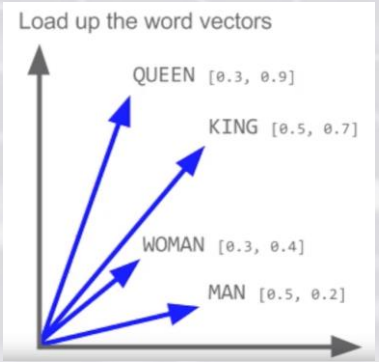
Word2vec (2013)

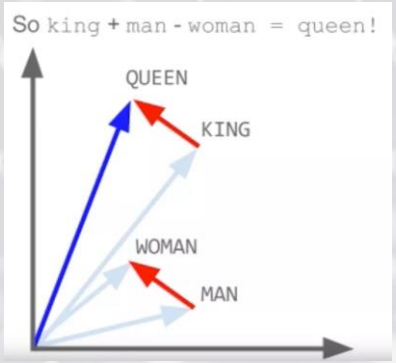
- **Word2vec** is a group of related models (Google) that are used to produce word embeddings.

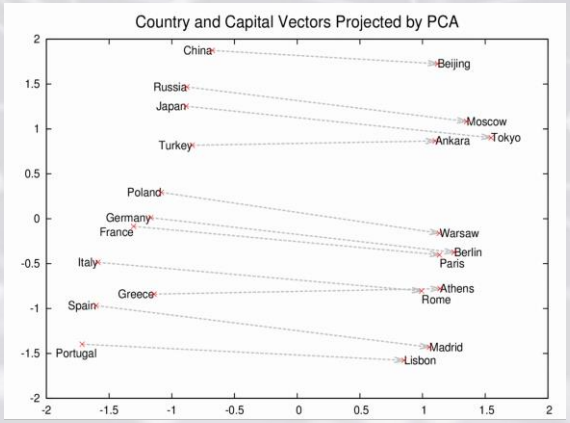
- These models are shallow, two-layer neural network that are trained to reconstruct linguistic contexts of words.

- Word2vec takes as its input a large corpus of text and produces a vector space (usually of high dimensions), with each unique word in the corpus being assigned a corresponding vector in the space.

- Word vectors are positioned in the vector space such that **words that share common contexts in the corpus are located in close proximity to one another in the space**.







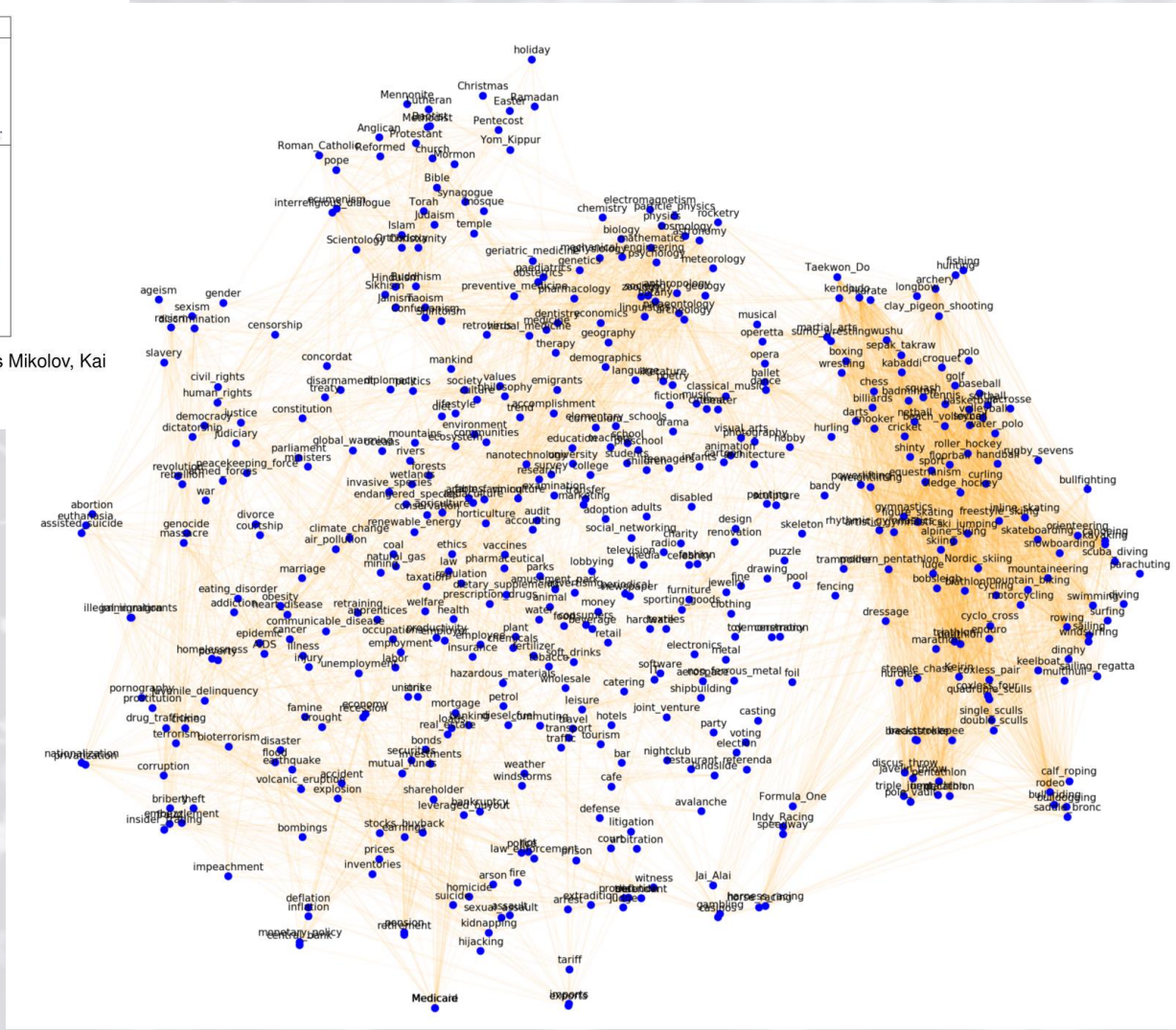
t-SNE for word2vec

Relations Learned by Word2vec

Word2vec model computed from 6 billion word corpus of news articles

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

"Efficient Estimation of Word Representations in Vector Space" Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, Arxiv 2013



Self-Organizing Maps (SOMs)

• To date, we have only considered applications of NNs for supervised learning, however, there exist several applications of NNs for unsupervised learning, including **self-organizing maps** (SOMs, 1988, Kohonen).

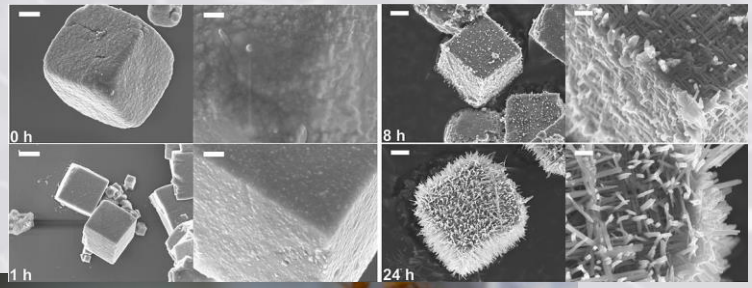
• In the unsupervised setting (e.g., k-means), we wish to identify meaningful data patterns in a *self-organizing fashion* (viz., without the use of labels). This process is often referred to as learning a **feature map** – that is to say, a compression scheme that illuminates structurally significant input features.

• Stated concisely, SOMs provide a way of performing dimensionality reduction using vector quantization. Furthermore, SOMs are unique in that they preserve topographic network properties that mimic biological processes in the brain.

Self-Organization & Complex Systems

(*) **Self-organization** is a process where some form of overall order arises from local interactions between parts of an initially disordered system. The process is spontaneous, not needing control by any external agent. It is often triggered by random fluctuations, amplified by positive feedback. The resulting organization is wholly decentralized, distributed over all the components of the system. As such, the organization is typically robust and able to survive or self-repair substantial perturbation.

Self-organization occurs in many physical, chemical, biological, robotic, and cognitive systems. Systems formed from self-organization processes often exhibit **emergent behavior**.



Recommended reading: M. Mitchell, *Complexity: A Guided Tour*.

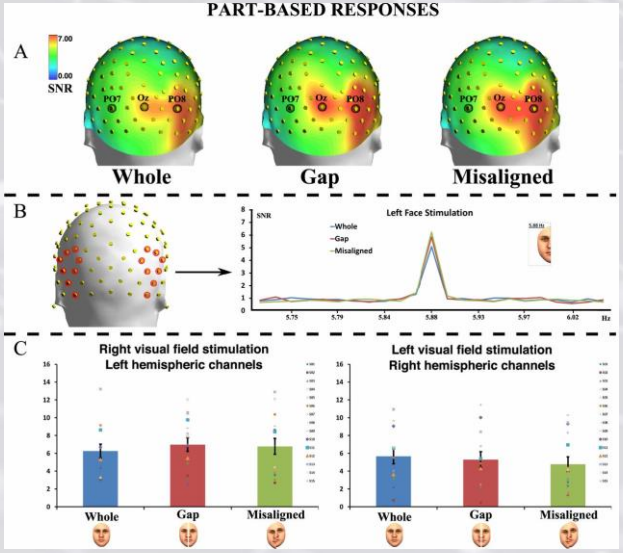
Topographic Maps & The Brain

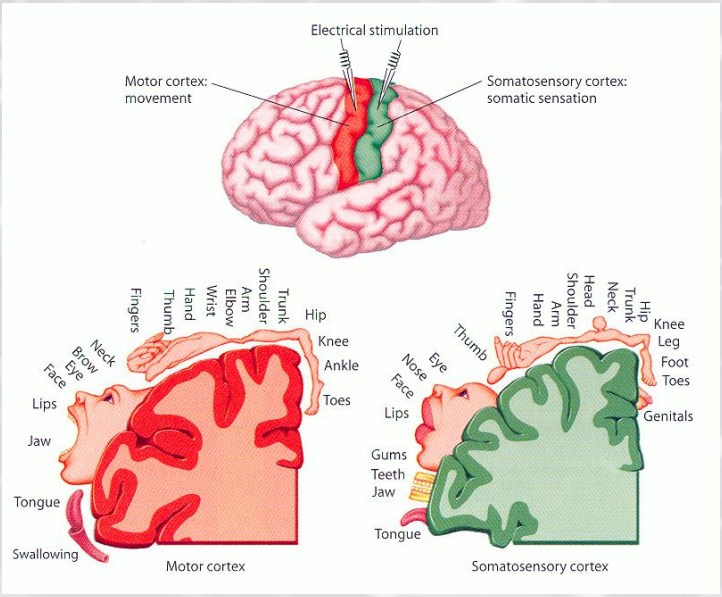
• Neurobiological studies indicate that different sensory inputs (motor, visual, auditory, etc.) are mapped onto corresponding areas of the cerebral cortex in an **orderly fashion**. This form of map, known as a **topographic map** has (2) important properties:

(1) Each piece of information is kept in its proper context/neighborhood;

(2) neurons dealing with closely-related pieces of information are kept close together so that they can interact using short synaptic connections.

(*) <u>SOMs train an artificial topographic map through self-organization in a neurobiologically inspired way</u>, abiding by the **principle of topographic map formation**: "The spatial location of an output neuron in a topographic map corresponds to a particular domain or feature drawn from the input space."





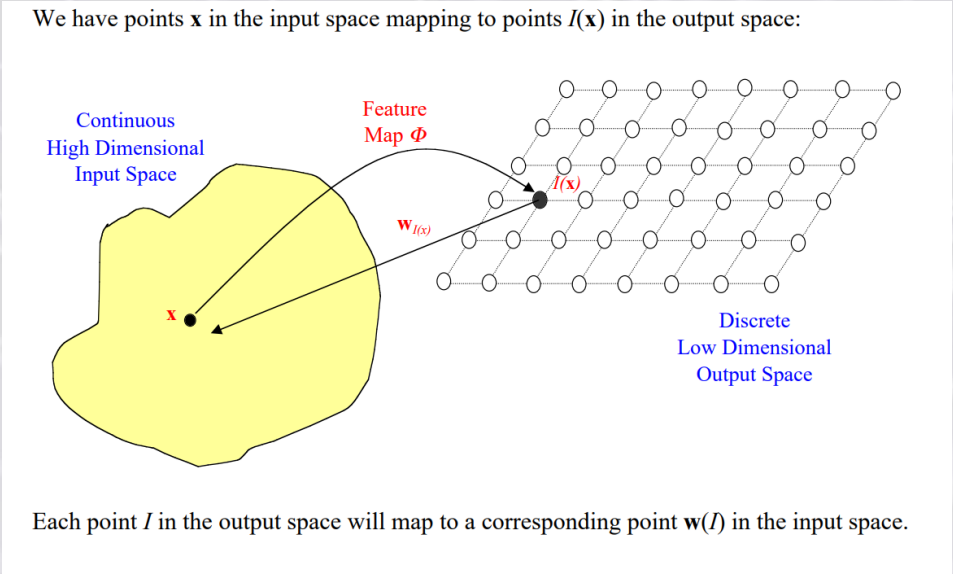
http://jov.arvojournals.org/article.aspx?articleid=2121362

SOMs: Overview

• The goal of a SOM is to transform the incoming signal pattern into a lower dimensional discrete map, and to perform this transformation adaptively in a topographically-ordered fashion (so that neurons that are close together represent inputs that are close together, while neurons that are far apart represent inputs that are far apart).

• SOMs utilize a class of unsupervised learning techniques known as **competitive learning**, in which output neurons compete amongst themselves to be activated, with the result being that only one is activated for a given input.

• This activated neuron is called a **winner-takes-all neuron** (also: **winning neuron**). Neurons become selectively tuned to various input patterns during the course of competitive learning.

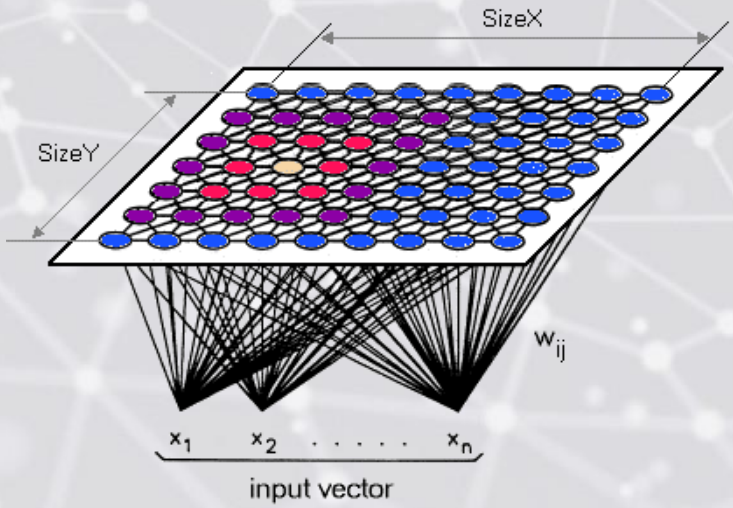


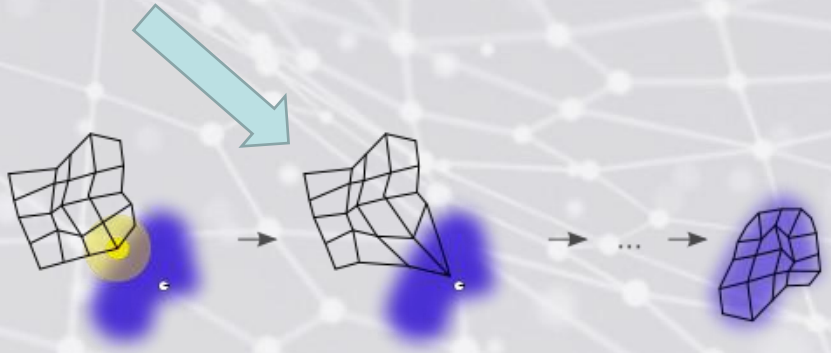
SOMs

• Note that with SOMs, the relative locations of the neurons in the network matters (nearby neuron correspond to similar input patterns) and the **neurons are arranged in a lattice/grid** (usually in 1-D or 2-D) with connections between the neurons, rather than in layers with connections only between different layers (as with the previous NNs we've seen). Each neuron is fully connected to all the source nodes in the input layer.

• Each node has a specific topological position (an (x,y) coordinate in the lattice) and contains a vector of weights.

• For training, neurons are tuned to conform with the topographic map criteria; in this way, the winning neuron should pull other neurons that are close to it in the network closer to itself in weight space, whereas neurons that are very far away should be ignored.





SOM Algorithm

The Self-Organising Feature Map Algorithm

- **Initialisation**
  - choose a size (number of neurons) and number of dimensions d for the map
  - Either:
    - choose random values for the weight vectors so that they are all different OR
    - set the weight values to increase in the direction of the first d principal components of the dataset
- **Learning**
  - repeat:
    - for each datapoint:
      - select the best-matching neuron n_b using the minimum Euclidean distance between the weights and the input,

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\|. \tag{9.8}$$

    - update the weight vector of the best-matching node using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)(\mathbf{x} - \mathbf{w}_j^T), \tag{9.9}$$

      where $\eta(t)$ is the learning rate.
    - update the weight vector of all other neurons using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta_n(t)h(n_b, t)(\mathbf{x} - \mathbf{w}_j^T), \tag{9.10}$$

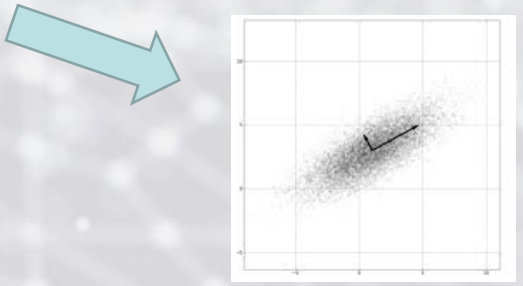
      where $\eta_n(t)$ is the learning rate for neighbourhood nodes, and $h(n_b, t)$ is the neighbourhood function, which decides whether each neuron should be included in the neighbourhood of the winning neuron (so $h = 1$ for neighbours and $h = 0$ for non-neighbours)
    - reduce the learning rates and adjust the neighbourhood function, typically by $\eta(t+1) = \alpha\eta(t)^{k/k_{max}}$ where $0 \leq \alpha \leq 1$ decides how fast the size decreases, k is the number of iterations the algorithm has been running for, and $k_{\max}$ is when you want the learning to stop. The same equation is used for both learning rates (η, η_n) and the neighbourhood function $h(n_b, t)$.
  - until the map stops changing or some maximum number of iterations is exceeded
- **Usage**
  - for each test point:
    - select the best-matching neuron n_b using the minimum Euclidean distance between the weights and the input:

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\| \tag{9.11}$$

SOM Algorithm: Overview

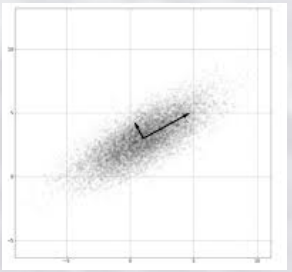
(I) **Initialization**: network parameters: determine number of neurons, dimension for the map (d)

-- can use a random initialization or begin with, say the PCA algorithm, using first *d* principal components.



SOM Algorithm: Overview

(I) **Initialization**: network parameters: determine number of neurons, dimension for the map (d)
-- can use a random initialization or begin with, say the PCA algorithm, using first *d* principal components.



(II) **Learning**:

(a) For each data point, *select best-matching neuron* (n_b), using minimum Euclidean distance.

(b) Update weight vector of n_b: $\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)\left(\mathbf{x} - \mathbf{w}_j^T\right)$
(this update has the effect of moving the weight vector of n_b closer to the datum), the learning rate $\eta(t)$ is decreased over time.

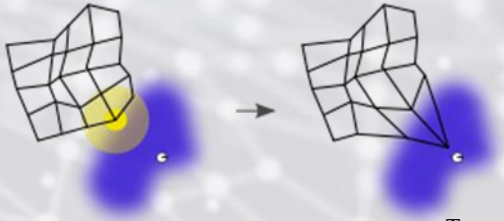
SOM Algorithm: Recap

(II) **Learning**:

(a) For each data point, *select best-matching neuron* (n_b), using minimum Euclidean distance.

(b) Update weight vector of n_b: $\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)\left(\mathbf{x} - \mathbf{w}_j^T\right)$

(this update has the effect of moving the weight vector of n_b closer to the datum), the learning rate $\eta(t)$ is decreased over time.



(c) Update the weight vector of <u>all other neurons</u> using: $\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta_n(t)h(n_b,t)\left(\mathbf{x} - \mathbf{w}_j^T\right)$

where $\eta_n(t)$ is the learning rate for the neighborhood nodes, and $h(n_b,t)$ is the **neighborhood function** with respect to node n_b, which decides whether each neuron should be included in the neighborhood of the winning neuron (e.g. n=1 for neighbors and n=0 for non-neighbors – or a Gaussian function can be used).

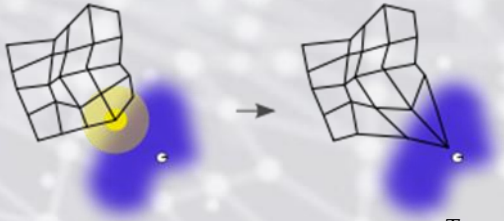
SOM Algorithm: Recap

(II) **Learning**:

(a) For each data point, *select best-matching neuron* (n_b), using minimum Euclidean distance.

(b) Update weight vector of n_b: $\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)\left(\mathbf{x} - \mathbf{w}_j^T\right)$

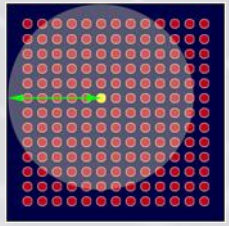
(this update has the effect of moving the weight vector of n_b closer to the datum), the learning rate $\eta(t)$ is decreased over time.



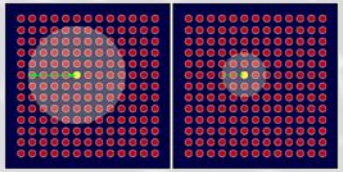
(c) Update the weight vector of all other neurons using: $\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta_n(t)h(n_b,t)\left(\mathbf{x} - \mathbf{w}_j^T\right)$

where $\eta_n(t)$ is the learning rate for the neighborhood nodes, and $h(n_b,t)$ is the **neighborhood function** with respect to node n_b, which decides whether each neuron should be included in the neighborhood of the winning neuron (e.g. n=1 for neighbors and n=0 for non-neighbods – or a Gaussian function can be used).

(d) Reduce the learning rates and adjust the neighborhood function (neighborhood size decreases over time).



$h(n_b,t)$ function



neighboorhood size decreases over time

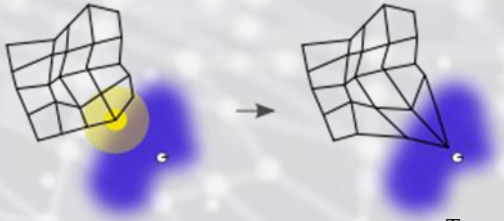
SOM Algorithm: Recap

(II) **Learning**:

(a) For each data point, *select best-matching neuron* (n_b), using minimum Euclidean distance.

(b) Update weight vector of n_b: $\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)\left(\mathbf{x} - \mathbf{w}_j^T\right)$

(this update has the effect of moving the weight vector of n_b closer to the datum), the learning rate $\eta(t)$ is decreased over time.



(c) Update the weight vector of all other neurons using: $\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta_n(t)h(n_b,t)\left(\mathbf{x} - \mathbf{w}_j^T\right)$

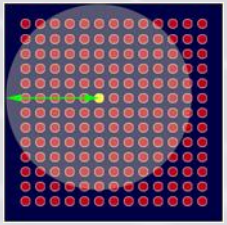
where $\eta_n(t)$ is the learning rate for the neighborhood nodes, and $h(n_b,t)$ is the **neighborhood function** with respect to node n_b, which decides whether each neuron should be included in the neighborhood of the winning neuron (e.g. n=1 for neighbors and n=0 for non-neighbods – or a Gaussian function can be used).

(d) Reduce the learning rates and adjust the neighborhood function (neighborhood size decreases over time).

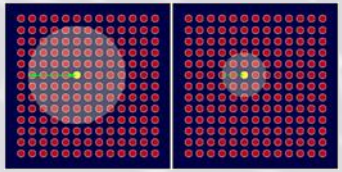
(III) **Testing**:

For each test point select best-matching neuron:

$$n_b = \min_j \left\| \mathbf{x} - \mathbf{w}_j^T \right\|$$

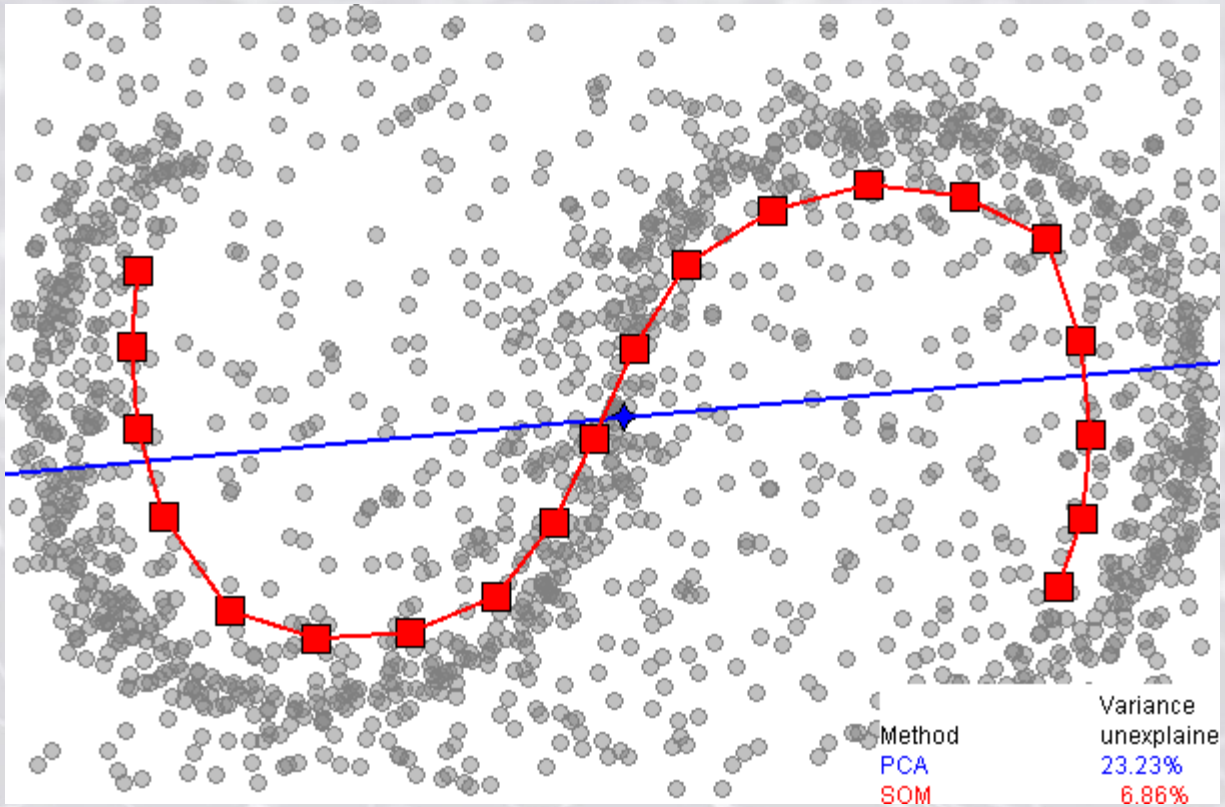


h(n_b,t) function



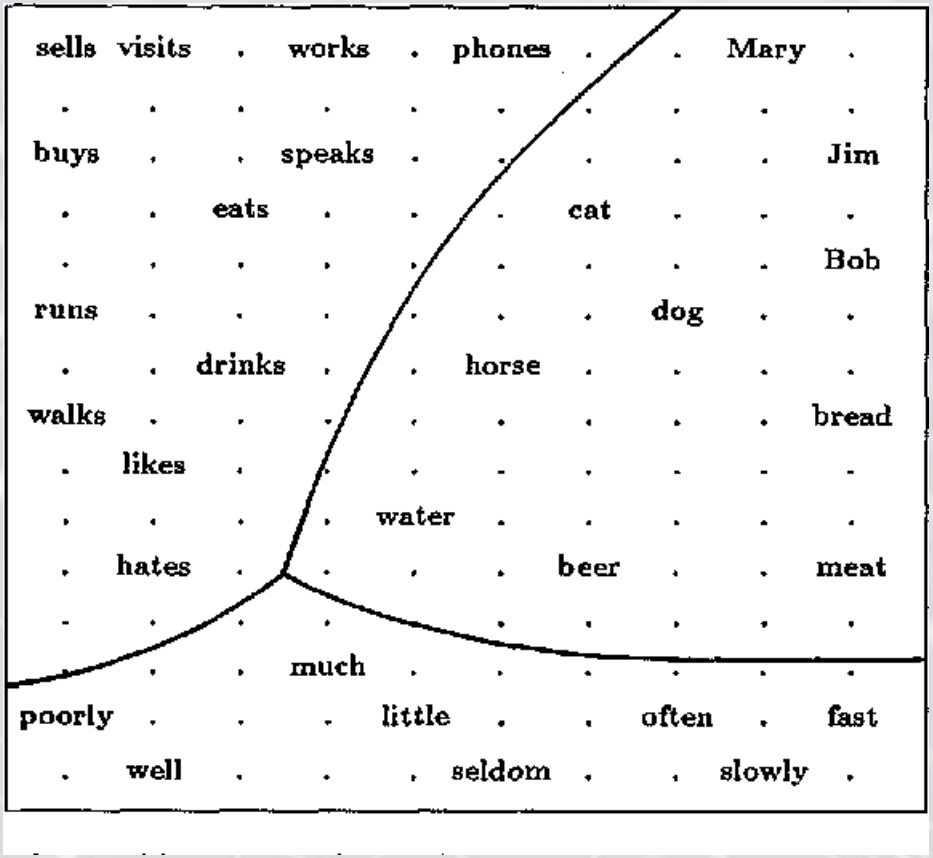
neighboorhood size decreases over time

SOM vs PCA



One-dimensional SOM versus principal component analysis (PCA) for data approximation. SOM is a red broken line with squares, 20 nodes. The first principal component is presented by a blue line. Data points are the small grey circles. For PCA, the fraction of variance unexplained in this example is 23.23%, for SOM it is 6.86%.

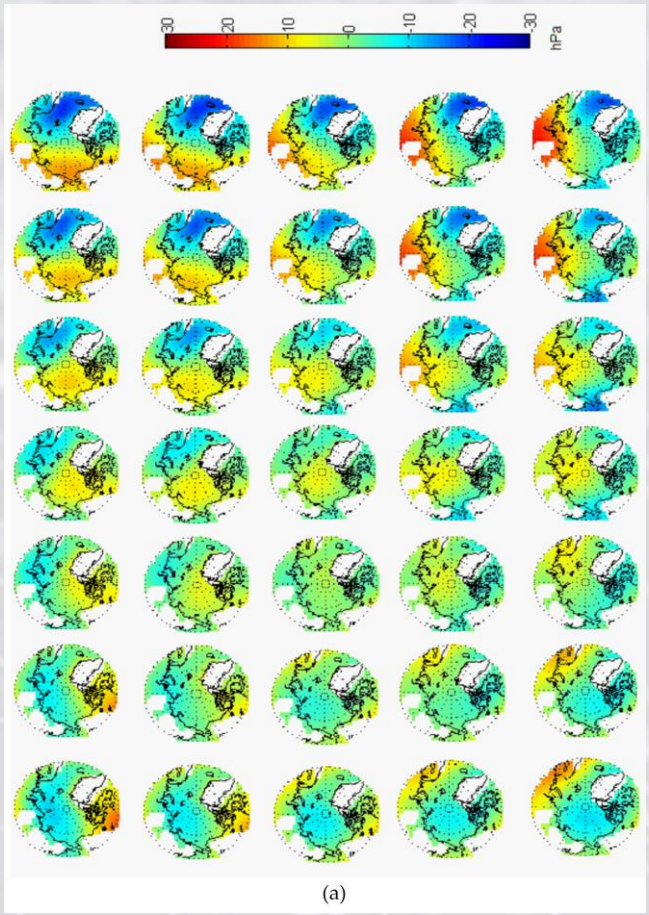
SOM for Semantic Maps



Semantic network (SOM) detects "logical similarity" between words based on statistics of their contexts (e.g. word order).

https://www.semanticscholar.org/paper/Self-organizing-semantic-maps-Ritter-Kohonen/7e6429291b65b4984a461350f7a07a3af1af7029

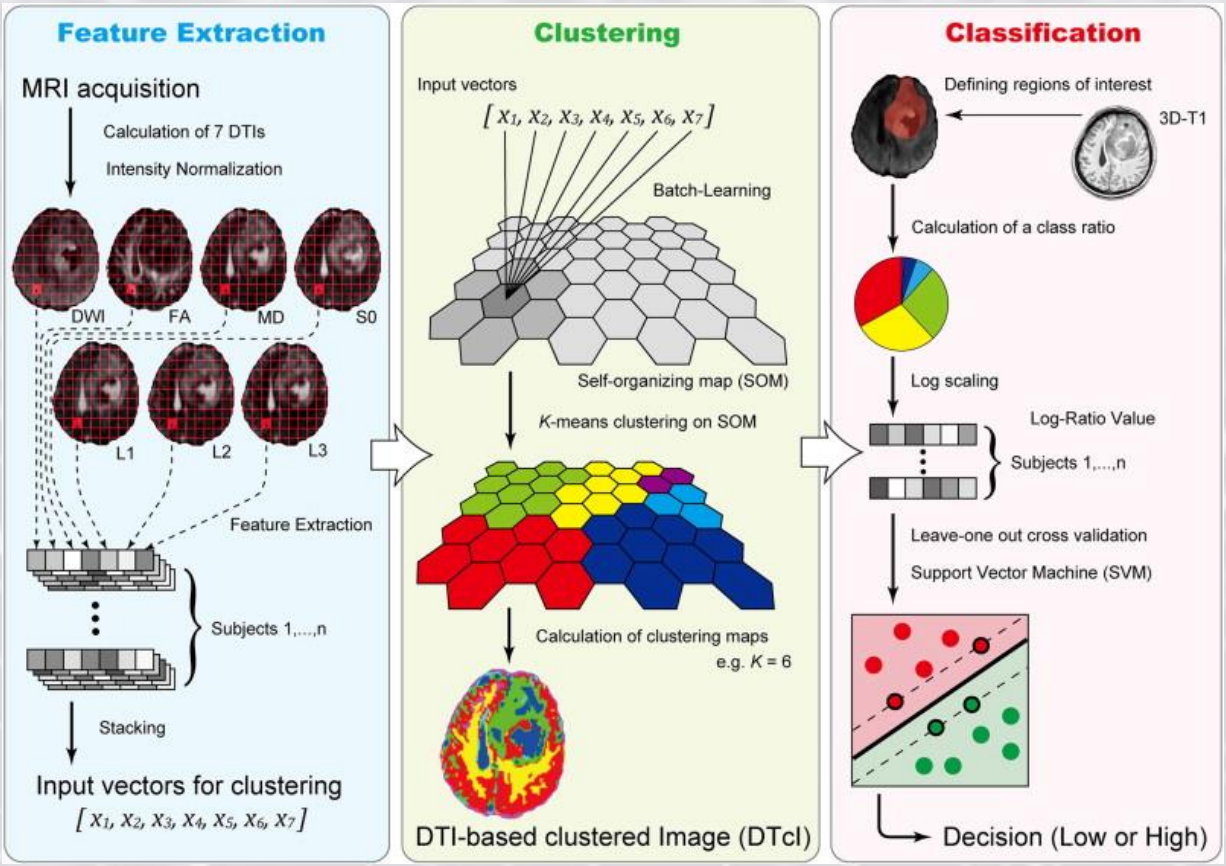
SOM for Atmospheric Science



SOM of sea level pressure anomaly patterns; different days fall into different categories, allowing researchers to attribute causes for variation with greater specificity.

https://www.intechopen.com/books/applications-of-self-organizing-maps/self-organizing-maps-a-powerful-tool-for-the-atmospheric-sciences

SOM for Medical Diagnosis



Pipeline used to predict glioma (tumor) grade and subsequently guide therapeutic strategies. First MRI data is acquired, the data was clustered in (2) steps beginning with an SOM, followed by k-means; lastly classification between high and low gliomas was done using an SVM.

https://www.nature.com/articles/srep30344

Dimensionality Reduction with Autoencoders

- Hinton *et al.*, devised a non-linear generalization of PCA that uses adaptive, multilayer "encoder" networks to transform high-dimensional data into a low-dimensional code and a similar "decoder" network to recover the data from the code.

https://www.cs.toronto.edu/~hinton/science.pdf

Dimensionality Reduction with Autoencoders

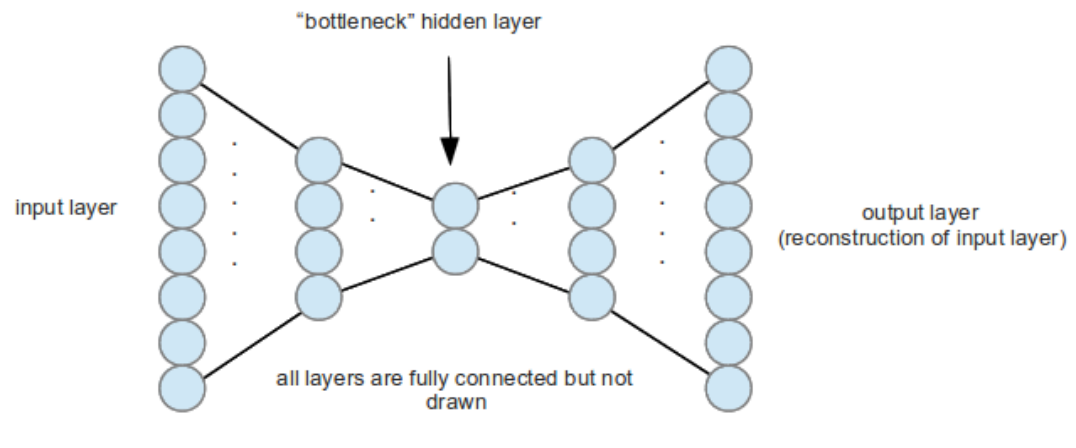
- Note that PCA is intimately connected with MLPs.

(*) An MLP can perform (non-linear) PCA using what is called an *auto-associator* (more commonly: **auto-encoder**).

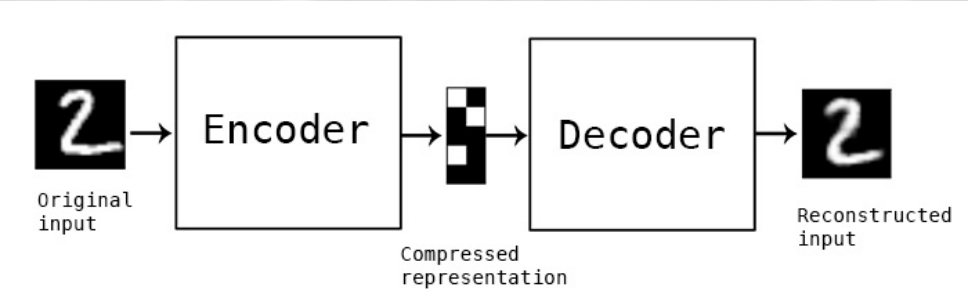
(*) If we train the MLP **where the output equals the input**, we are asking the network to learn a data "reconstruction" process; we therefore train to *minimize the reconstruction error*.

(*) Usually the hidden layers are smaller in dimension than the output/input layers so that they form a **compression "bottleneck"**.

(*) The activations at the hidden layers (i.e. the feature vectors) Encode a *dimensionality reduction* of the data.



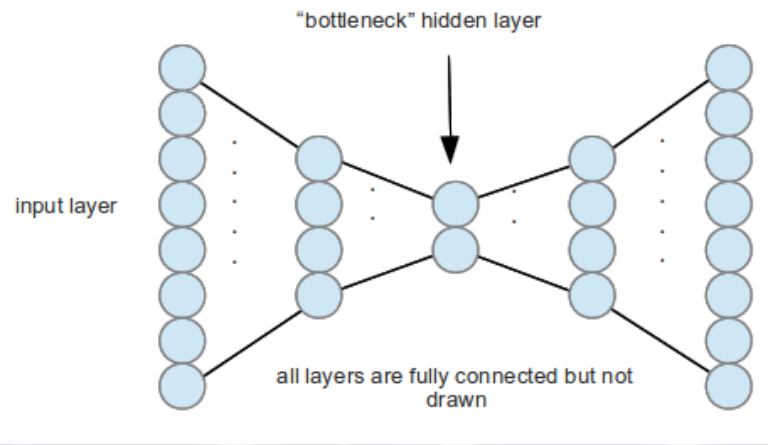
PCA & Auto-encoders: Image Denoising



The image shows how a "denoising" autoencoder may be used to generate correct input from corrupted input.

Dimensionality Reduction with Autoencoders

• It is difficult to optimize the weights in nonlinear autoencoders that have multiple hidden layers (2–4). With large initial weights, autoencoders typically find poor local minima; with small initial weights, the gradients in the early layers are tiny, making it infeasible to train autoencoders with many hidden layers.

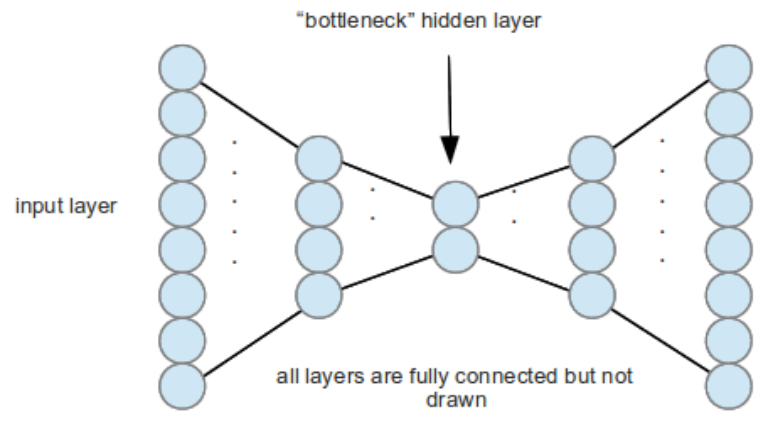


Dimensionality Reduction with Autoencoders

• It is difficult to optimize the weights in nonlinear autoencoders that have multiple hidden layers (2–4). With large initial weights, autoencoders typically find poor local minima; with small initial weights, the gradients in the early layers are tiny, making it infeasible to train autoencoders with many hidden layers.

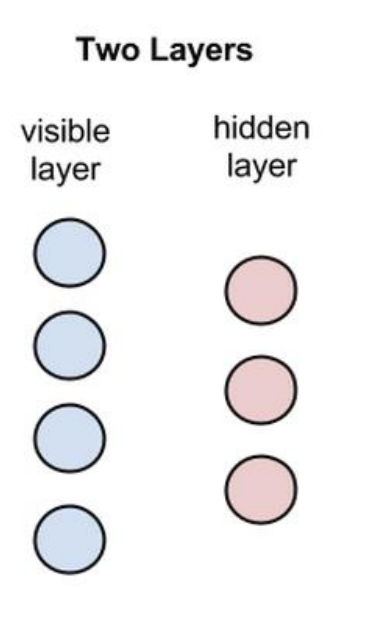
• If the initial weights are close to a good solution, gradient descent works well, but finding such initial weights requires a very different type of algorithm that learns one layer of features at a time.

• Hinton *et al.* introduce a pretraining procedure for binary data, and generalize it to real-valued data.



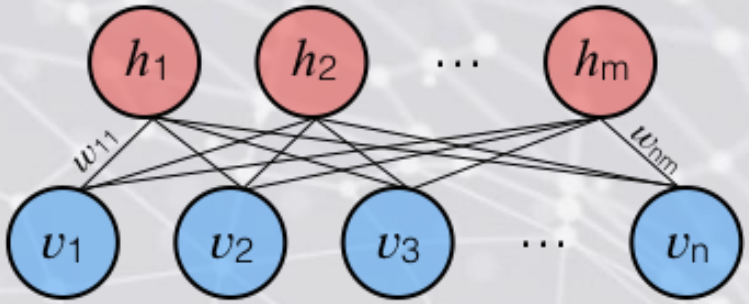
Aside: Restricted Boltzmann Machines (RBMs)

• RBMs are shallow, two-layer neural nets that constitute the building blocks of *deep-belief networks (the layers of which can act as feature detectors)*. The first layer of the RBM is called the visible, or input, layer, and the second is the hidden layer.



Aside: Restricted Boltzmann Machines (RBMs)

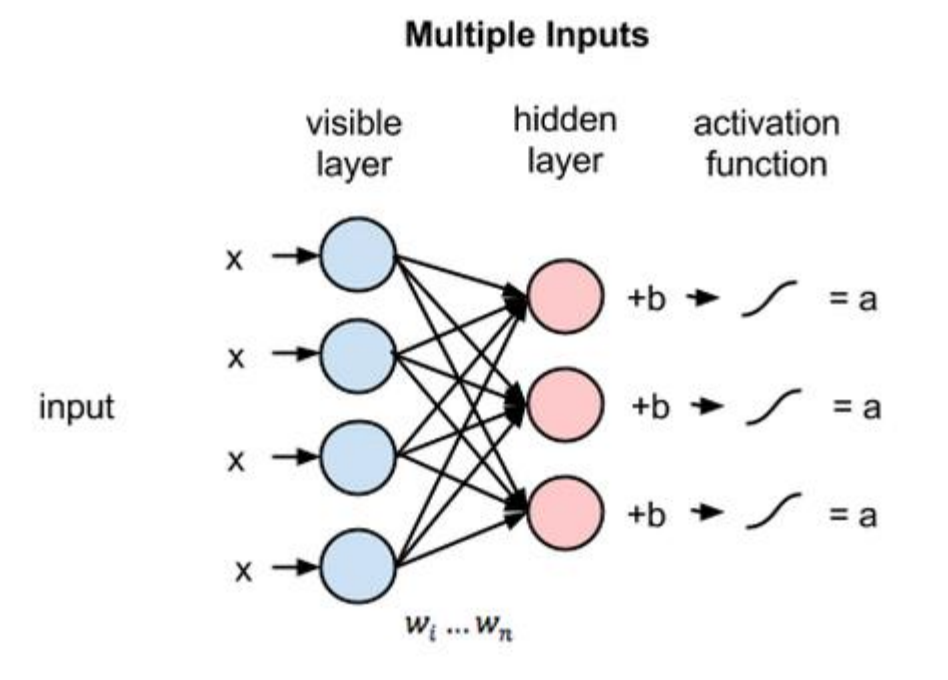
• RBMs are shallow, two-layer neural nets that constitute the building blocks of *deep-belief networks (the layers of which can act as feature detectors)*. The first layer of the RBM is called the visible, or input, layer, and the second is the hidden layer.



• The nodes are fully connected across the layers – but there are no intra-layer connections (this is the indicated restriction in the RBM); the underlying graph is, in other words, **bipartite**.

Aside: Restricted Boltzmann Machines (RBMs)

• Each visible node takes a feature from the input (just as with perceptrons, NNs, etc.)

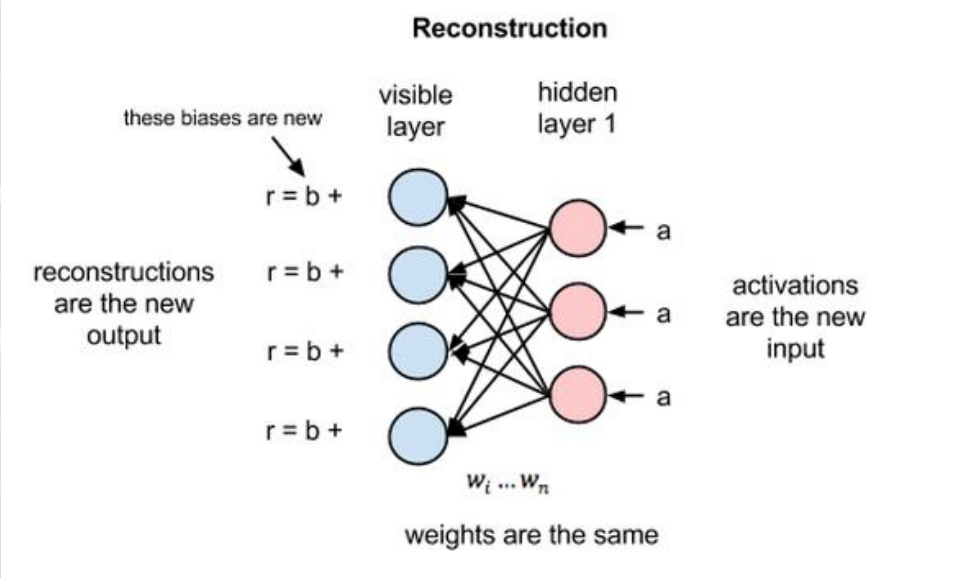


• As usual, at each hidden node, each input x is multiplied by its respective weight w. Each hidden node receives the inputs multiplied by their respective weights. The sum of those products is again added to a bias, and the result is passed through the activation algorithm producing one output for each hidden node.

Aside: Restricted Boltzmann Machines (RBMs)

• For *reconstruction*, the activations in the hidden layer now become the input for backward phase.

• They are multiplied by the same weights, one per internode edge, just as x was weight-adjusted on the forward pass. The sum of those products is added to a visible-layer bias at each visible node, and the output of those operations is a **reconstruction**; i.e. an approximation of the original input.

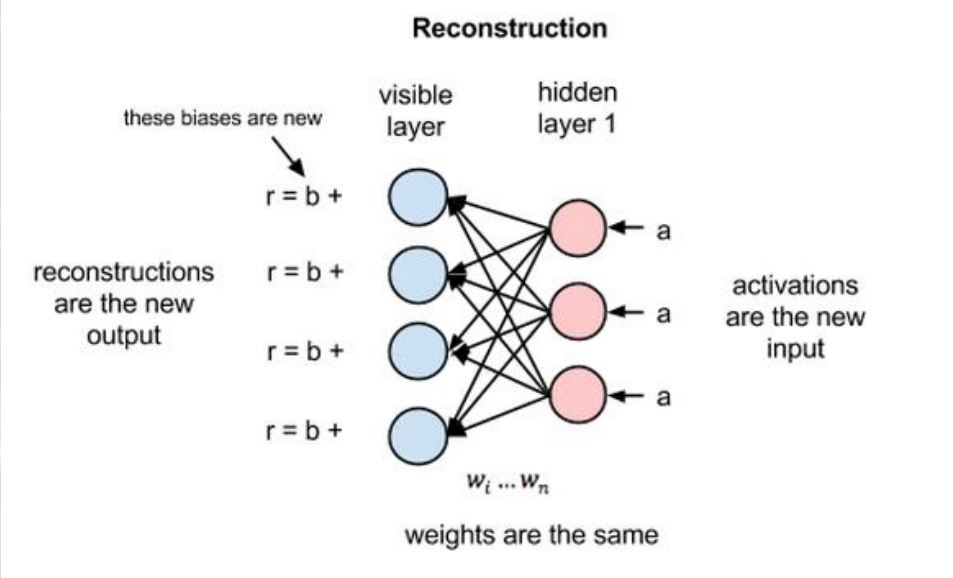


Aside: Restricted Boltzmann Machines (RBMs)

• For the forward phase we can think of the RBM as using the inputs to make predictions about node activations, i.e. P(a|x,w).

•Conversely, in the backward phase, because the activations are fed in and reconstruction approximations are outputted, this phase can be as summarized as approximating the distribution: P(x|a,w).

• Together, then, these two estimates together represent the joint distribution: P(x,a|w).

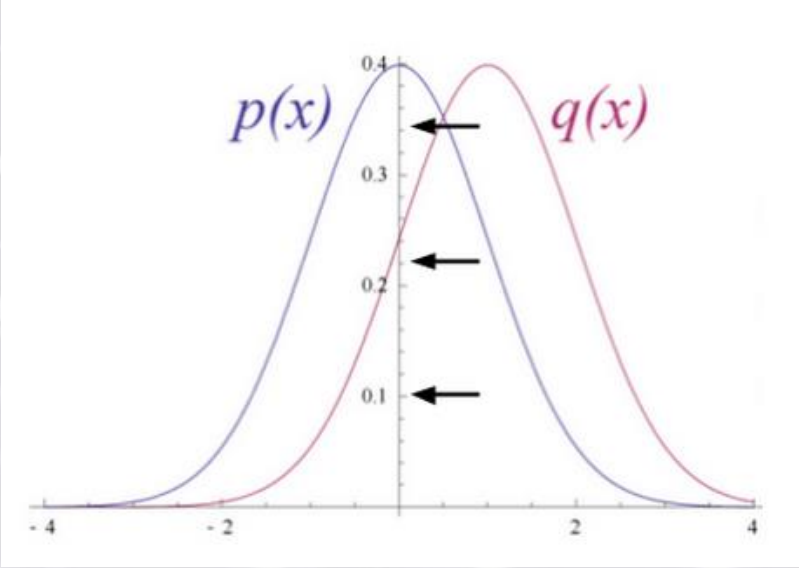


Aside: Restricted Boltzmann Machines (RBMs)

• The reconstruction process is making guesses about the distribution of the input. Consequently, reconstruction is an instance of generative learning.

• Since RBMs are learning to reconstruct the input, we can think of an RBM as minimizing the "difference" between the input (distribution) and reconstruction (distribution).

• Put another way – the RBM works to minimize the KL divergence of these two distributions.



Aside: Restricted Boltzmann Machines (RBMs)

• The process of learning reconstructions is, in a sense, learning which groups of pixels tend to co-occur for a given set of images.

•If, say, an RBM were only fed images of dogs and cats (with only two output nodes, one for each). The question the RBM is asking itself on the forward pass is: Given these pixels, should my weights send a stronger signal to the dog node or the cat node? And the question the RBM asks on the backward pass is: Given, say, a dog, which distribution of pixels should I expect?

Aside: Restricted Boltzmann Machines (RBMs)

• The process of learning reconstructions is, in a sense, learning which groups of pixels tend to co-occur for a given set of images.

•If, say, an RBM were only fed images of dogs and cats (with only two output nodes, one for each). The question the RBM is asking itself on the forward pass is: Given these pixels, should my weights send a stronger signal to the dog node or the cat node? And the question the RBM asks on the backward pass is: Given, say, a dog, which distribution of pixels should I expect?

• Note that RBMs have two biases. This is one aspect that distinguishes them from other autoencoders. The hidden bias helps the RBM produce the activations on the forward pass (since biases impose a floor so that at least some nodes fire no matter how sparse the data), while the bias in the *visible* nodes helps the RBM learn the reconstructions on the backward pass.

Aside: Restricted Boltzmann Machines (RBMs)

- Once the RBM learns the structure of the input data as it relates to the activations of the first hidden layer, then the data is passed one layer down the net.

- The first hidden layer takes on the role of visible layer. The activations now effectively become your input, and they are multiplied by weights at the nodes of the second hidden layer, to produce another set of activations.

Aside: Restricted Boltzmann Machines (RBMs)

• Once the RBM learns the structure of the input data as it relates to the activations of the first hidden layer, then the data is passed one layer down the net.

• The first hidden layer takes on the role of visible layer. The activations now effectively become your input, and they are multiplied by weights at the nodes of the second hidden layer, to produce another set of activations.

• This process of creating sequential sets of activations by grouping features and then grouping groups of features is the basis of a *feature hierarchy*, by which neural networks learn more complex and abstract representations of data.

• With each new hidden layer, the weights are adjusted until that layer is able to approximate the input from the previous layer. This is greedy, layer-wise and unsupervised pre-training. It requires no labels to improve the weights of the network, which means you can train on unlabeled data, untouched by human hands, which is the vast majority of data in the world.

Aside: Restricted Boltzmann Machines (RBMs)

• RBMs are one example of so-called **energy-based models** in ML (e.g. *simulated annealing, Ising model*) – meaning that they are (loosely) analogous to physical systems for which **"stable" states** (sometimes called *basins of attraction/attractors*) represent low-energy configurations of the system.

•RBMS are typically trained with an algorithm called **Contrastive Divergence*** (details omitted here for brevity).

• Once an RBM is trained, another RBM is "stacked" atop it, taking its input from the final trained layer. This forms a **Deep Belief Network** DBN). The fact that DBNs can be trained greedily, one layer at a time, led to one of the first effective deep learning algorithms.**

*http://www.cs.toronto.edu/~fritz/absps/cdmiguel.pdf
**http://www.iro.umontreal.ca/~lisa/pointeurs/TR1312.pdf

Aside: Restricted Boltzmann Machines (RBMs)

• One can sample from a trained RBM in order to generate reconstructed samples – Hinton equated this with the algorithm "dreaming."

• More formally, **Gibbs sampling** can be used to generate these samples by beginning with the hidden layer (for some initial, random string), sample the visible layer, generating a new string and then repeat this process for many iterations.

Aside: Restricted Boltzmann Machines (RBMs)

Human names

Here are some samples drawn from a model trained on the full names of 1.5m actors from IMDB (more here):

```
omar vole
r.j. pen
ronald w. males
jean-paul recan
marxel sode
samuel j. varga
lionel cone
```

Some examples from an order-4 model trained on the US place names dataset:

```
Bonny Maringer City of Lake
Sour Motoruk Mountain
Mount Branchorage Lakes
Duck Kill Bar Rock
Goatyard Point
Noblit Hollow
Spenceton
Jay Canal Cemetery
Oriflamming Beach
```

Dimensionality Reduction with NNs*

• An ensemble of binary vectors (e.g., images) can be modeled using a two-layer RBM in which stochastic, binary pixels are connected to stochastic, binary feature detectors using symmetrically weighted connections.

• The pixels correspond to "visible" units of the RBM because their states are observed; the feature detectors correspond to "hidden" units. A joint configuration (v, h) of the visible and hidden units has an *energy* given by:

$$E(\mathbf{v},\mathbf{h}) = -\sum_{i \in pixels} b_i v_i - \sum_{j \in features} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

where v_i and h_j are the binary states of pixel *i* and feature *j*, b_i and b_j are their biases, and w_{ij} is the weight between them. The network assigns a probability to every possible image via this energy function (as we explained previously).

• The probability of a training image can be raised by adjusting the weights and biases to lower the energy of that image and to raise the energy of similar, reconstructed images that the network would prefer to the real data.

*https://www.cs.toronto.edu/~hinton/science.pdf

Dimensionality Reduction with NNs

• Given a training image, the binary state h_j of each feature detector *j* is set to 1 with probability $\sigma(b_j + \sum_i v_i w_{ij})$, where $\sigma(x)$ is the logistic function, b_j is the bias of j, v_i is the state of pixel i, and w_{ij} is the weight between i and j.

•Once binary states have been chosen for the hidden units, a "confabulation" is produced by setting each v_i to 1 with probability $\sigma(b_j + j\sum h_j w_{ij})$, where b_i is the bias of i. The states of the hidden units are then updated once more so that they represent features of the confabulation. The change in a weight is given by:

$$\Delta w_{ij} = \eta\left(\left\langle v_i h_j \right\rangle_{data} - \left\langle v_i h_j \right\rangle_{reconstruction}\right)$$

where η is the learning rate, $\langle v_i h_j \rangle_{data}$ is the fraction of times that the pixel *i* and feature detector *j* are on together when the feature detectors are being driven by data, and $\langle v_i h_j \rangle_{reconstruction}$ is the corresponding fraction for confabulations. A simplified version of the same learning rule is used for the biases.

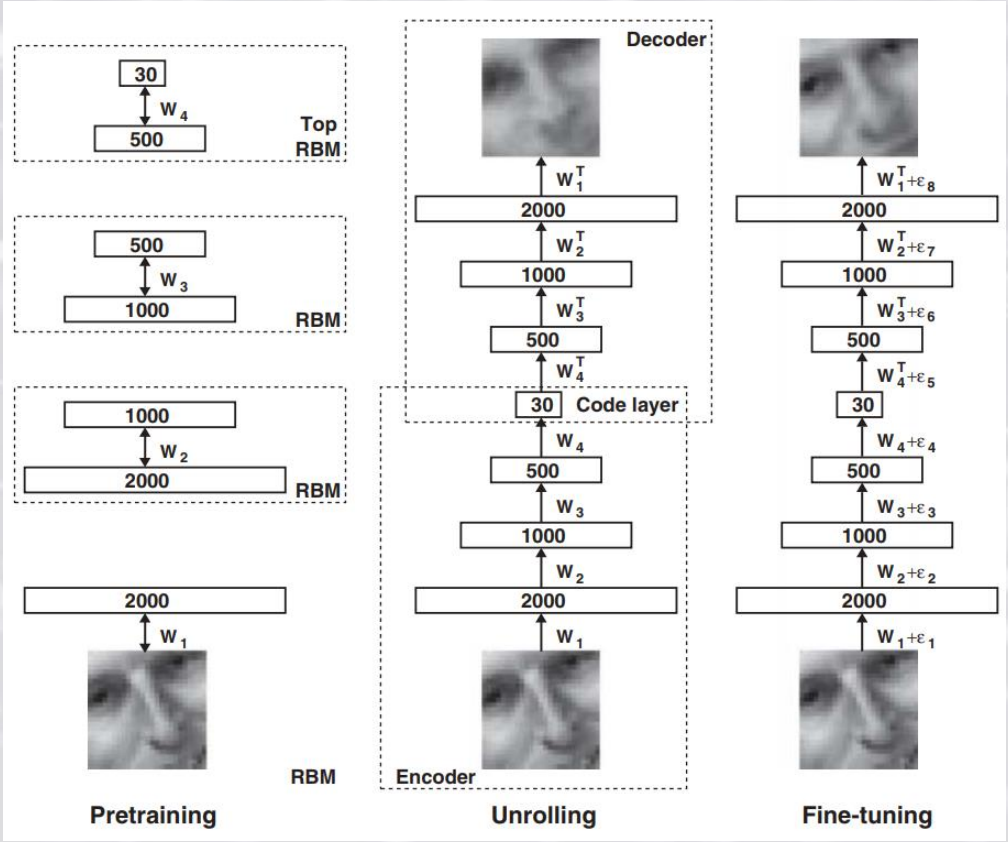
Dimensionality Reduction with NNs

• A single layer of binary features is not the best way to model the structure in a set of images. After learning one layer of feature detectors, we can treat their activities—when they are being driven by the data—as data for learning a second layer of features. The first layer of feature detectors then become the visible units for learning the next RBM, etc. This yields a stacked RBM model.

• Each layer of features captures strong, high-order correlations between the activities of units in the layer below. For a wide variety of data sets, this is an efficient way to progressively reveal low-dimensional, nonlinear structure.

Dimensionality Reduction with NNs

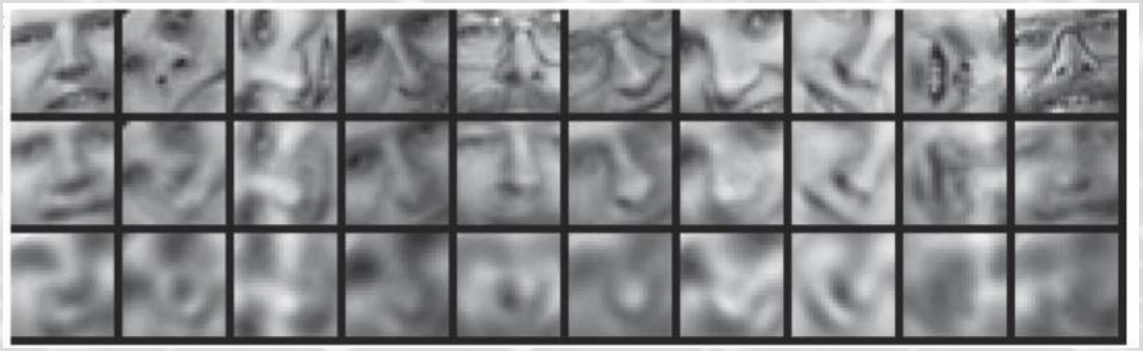
• After pretraining multiple layers of feature detectors, the model is "unrolled" to produce encoder and decoder networks that initially use the same weights. The global finetuning stage then replaces stochastic activities by deterministic, real-valued probabilities and uses backpropagation through the whole autoencoder to fine-tune the weights for optimal reconstruction.



Pretraining consists of learning a stack of restricted Boltzmann machines (RBMs), each having only one layer of feature detectors. The learned feature activations of one RBM are used as the "data" for training the next RBM in the stack. After the pretraining, the RBMs are "**unrolled**" to create a deep autoencoder, which is then fine-tuned using backpropagation of error derivatives.

Dimensionality Reduction with NNs

• Top to bottom: A random test image from each class; reconstructions by the 30-dimensional autoencoder; reconstructions by 30- dimensional logistic PCA and standard PCA. The average squared errors for the last three rows are 3.00, 8.01, and 13.87.



• Top to bottom: Random samples from the test data set; reconstructions by the 30-dimensional autoencoder; reconstructions by 30-dimensional PCA. The average squared errors are 126 and 135.

Dimensionality Reduction with NNs

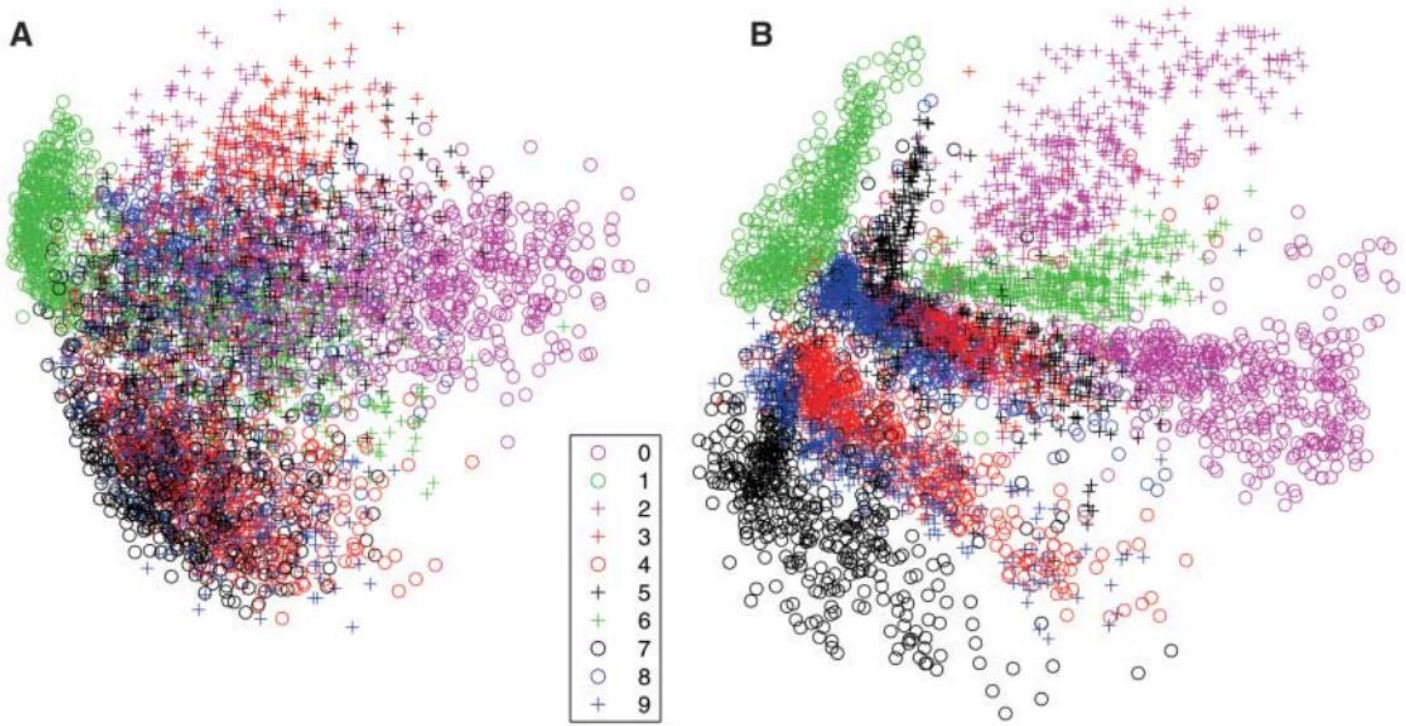
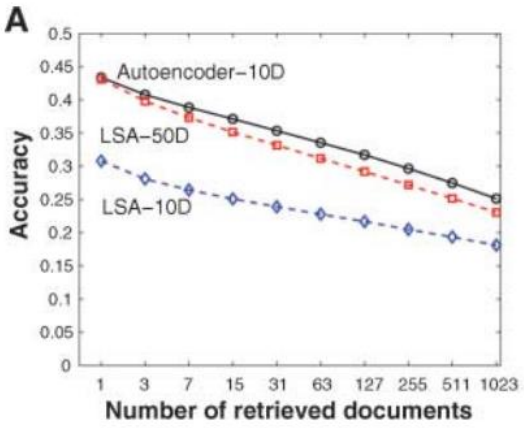
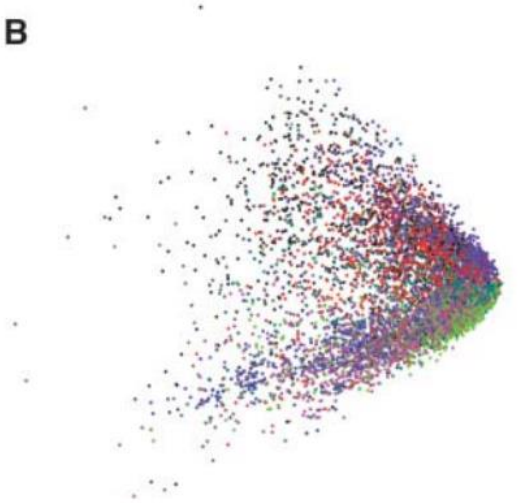


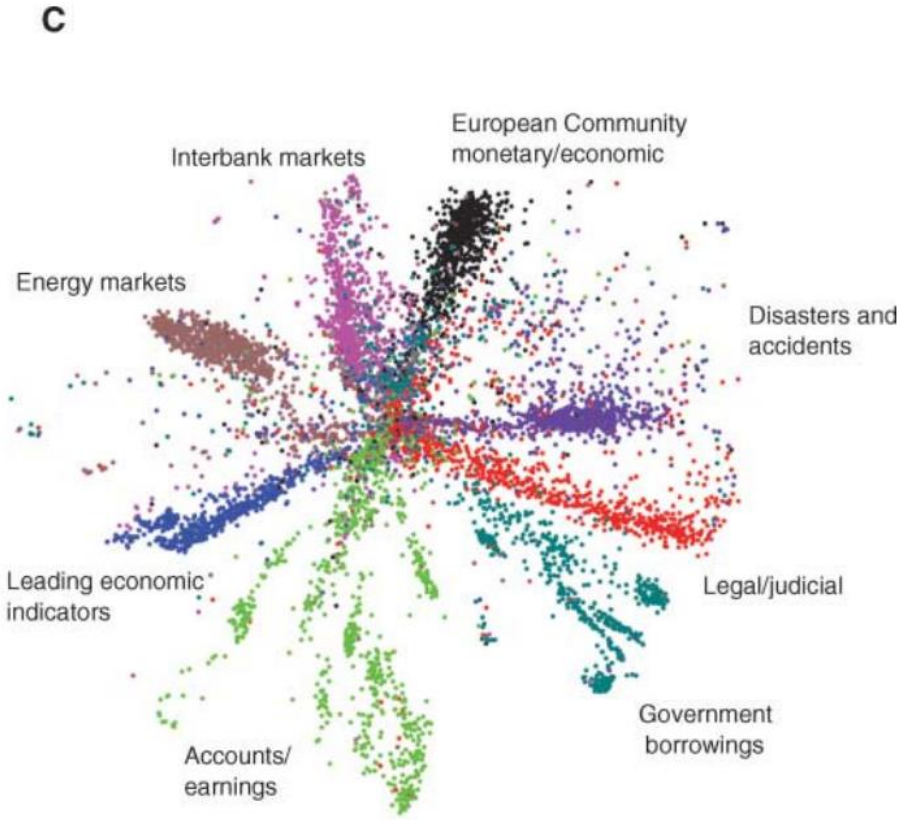
Fig. 3. (**A**) The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. (**B**) The two-dimensional codes found by a 784-1000-500-250-2 autoencoder. For an alternative visualization, see (*8*).

Dimensionality Reduction with NNs

Fig. 4. (**A**) The fraction of retrieved documents in the same class as the query when a query document from the test set is used to retrieve other test set documents, averaged over all 402,207 possible queries. (**B**) The codes produced by two-dimensional LSA. (**C**) The codes produced by a 2000-500-250-125-2 autoencoder.







Spectral Clustering

• **Spectral clustering** techniques make use of the spectrum (i.e. eigenvalues) of the similarity matrix of a data set to perform dimensionality reduction before before clustering in fewer dimensions. Spectral clustering is a non-linear dimensionality reduction scheme and can therefore represent a richer set of (low-dimensional) manifolds than linear dimensionality reduction schemes (e.g. PCA).

• The similarity matrix S (a symmetric matrix) is provided as an input and consists of a quantitative assessment of the relative similarity of each pair of points in the dataset.

Spectral Clustering

• The goal when constructing *similarity graphs* (NB: the similarity graph is simply the weighted undirected graph defined by its similarity matrix S) is to model the local neighborhood relationships between the data points. Moreover, most of the constructions below lead to a sparse representation of the data, which has computational advantages

Spectral Clustering

• The goal when constructing *similarity graphs* (NB: the similarity graph is simply the weighted undirected graph defined by its similarity matrix S) is to model the local neighborhood relationships between the data points. Moreover, most of the constructions below lead to a sparse representation of the data, which has computational advantages.

• Similarity can be defined according to different criteria; here are some of the most common criteria used in practice to define similarity:

The ε-neighborhood graph: Here we connect all points whose pairwise distances are smaller than ε. As the distances between all connected points are roughly of the same scale (at most ε), weighting the edges would not incorporate more information about the data to the graph. Hence, the ε-neighborhood graph is usually considered as an unweighted graph.

k-nearest neighbor graphs: Here the goal is to connect vertex v_i with vertex v_j if v_j is among the k nearest neighbors of v_i. However, this definition leads to a directed graph, as the neighborhood relationship is not symmetric. Now there are two ways of making this graph undirected. The first way is to simply ignore the directions of the edges, that is we connect v_i and v_j with an undirected edge if v_i is among the k-nearest neighbors of v_j *or* if v_j is among the k-nearest neighbors of v_i. The resulting graph is what is usually called *the k-nearest neighbor graph*. The second choice is to connect vertices v_i and v_j if both v_i is among the k-nearest neighbors of v_j *and* v_j is among the k-nearest neighbors of v_i. The resulting graph is called the *mutual k-nearest neighbor graph*. In both cases, after connecting the appropriate vertices we weight the edges by the similarity of the adjacent points.

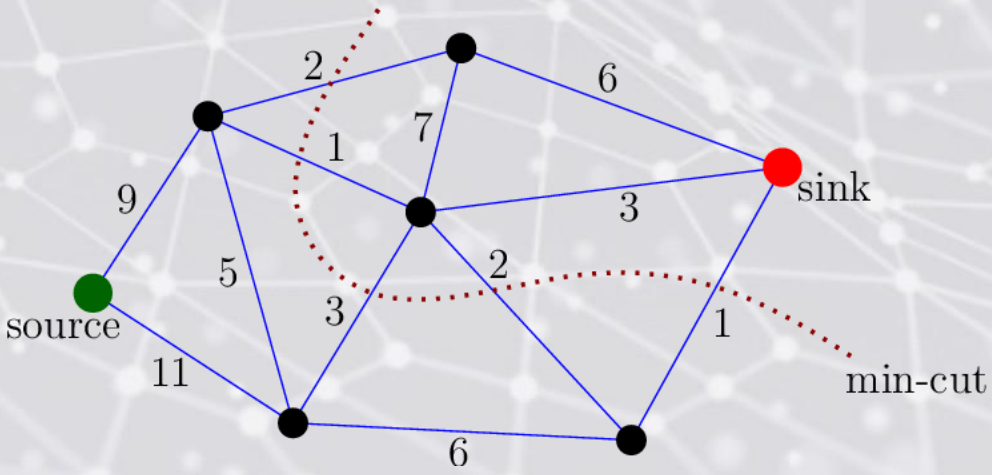
The fully connected graph: Here we simply connect all points with positive similarity with each other, and we weight all edges by s_{ij}. As the graph should model the local neighborhood relationships, this construction is usually only chosen if the similarity function itself already encodes mainly local neighborhoods. An example for a similarity function where this is the case is the Gaussian similarity function $s(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$. Here the parameter σ controls the width of the neighborhoods, similarly to the parameter ε in case of the ε-neighborhood graph.

Spectral Clustering

• One can consider clustering in terms of **graph cuts**. If we want to find a partition from the similarity graph for a data set into *K* clusters, say $A_1, \ldots, A_K$, a natural criterion is to minimize:

$$cut(A_1,\ldots,A_K)=\frac{1}{2}\sum_{k=1}^{K}W\left(A_k,\bar{A}_k\right)$$

where W denotes the similarity graph, $\overline{A_k}=V\backslash A_k$ is the *complement of* A_k, and $W(A,B)=\sum_{i\in A, j\in B} w_{ij}$. This criterion is, in other words, just the standard min-cut criterion.



Spectral Clustering

• One can consider clustering in terms of **graph cuts**. If we want to find a partition from the similarity graph for a data set into *K* clusters, say $A_1, \ldots, A_K$, a natural criterion is to minimize:

$$cut(A_1,\ldots,A_K)=\frac{1}{2}\sum_{k=1}^{K}W\left(A_k,\bar{A}_k\right)$$

where W denotes the similarity graph, $\overline{A_k}=V\backslash A_k$ is the *complement of* A_k, and $W(A,B)=\sum_{i\in A,j\in B} w_{ij}$. This criterion is, in other words, just the standard min-cut criterion.

More commonly, researchers use the **normalized cut criterion**, defined as:

$$Ncut(A_1,\ldots,A_K)=\frac{1}{2}\sum_{k=1}^{K}\frac{W\left(A_k,\bar{A}_k\right)}{vol\left(A_k\right)}$$

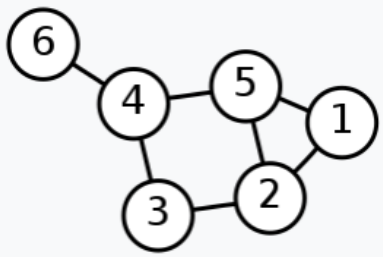
where $vol(A_k)=\sum_{i\in A} d_i$, where d_i is the weighted degree of vertex *i* in the similarity graph.

(*) This splits the graph into *K* clusters such that nodes within each cluster are similar to one another, but are different to nodes in other clusters.

Spectral Clustering: Graph Laplacian

• Let W be the symmetric weight matrix for a graph where $w_{ij}=w_{ji} \geq 0$. Let $D=\mathrm{diag}(d_i)$ be the diagonal matrix containing the weighted degree of each node.

The **graph Laplacian** is defined as: L = D – W

Labeled graph	Degree matrix	Adjacency matrix	Laplacian matrix
	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$

Spectral Clustering: Graph Laplacian

- The **graph Laplacian** is defined as: $L = D - W$.

The Laplacian possesses several useful properties, including the fact that it is symmetric and positive semi-definite.

(*) Consequently, L has non-negative, real-valued eigenvalues: $0 \leq \lambda_1 \leq \lambda_2 \ldots \leq \lambda_N$.
(this property holds by virtue of L being positive semi-definite)

How is this helpful for clustering?

Spectral Clustering: Graph Laplacian

- The **graph Laplacian** is defined as: $L = D - W$.

The Laplacian possesses several useful properties, including the fact that it is symmetric and positive semi-definite.

(*) Consequently, L has non-negative, real-valued eigenvalues: $0 \leq \lambda_1 \leq \lambda_2 \ldots \leq \lambda_N$.

How is this helpful for clustering?

A handy theorem states that the set of eigenvectors of L with eigenvalue 0 is spanned by the indicator vectors $\mathbf{1}_{A1},\ldots,\mathbf{1}_{AK}$, where A_k are the K connected components of the graph.

(*) Consequently, the **number of connected components of the graph** is given by the dimension of the nullspace of L (also called simply the "*nullity* of L" in linear algebra); put another way: the *algebraic multiplicity* of the zero eigenvalue equals the number of connected components of the graph.

Spectral Clustering: Graph Laplacian

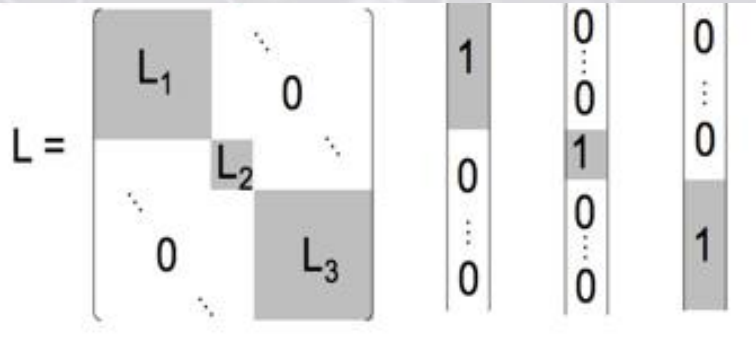
• The **graph Laplacian** is defined as: $L = D - W$.

(*) Consequently, L has non-negative, real-valued eigenvalues: $0 \leq \lambda_1 \leq \lambda_2 \ldots \leq \lambda_N$.

(*) Consequently, the **number of connected components of the graph** is given by the dimension of the nullspace of L; put another way: the *algebraic multiplicity* the of the zero eigenvalue equals the number of connected components of the graph.

(*) Thus, in the "ideal" case for which the data graph consists of *k* disconnected components, then the multiplicity of the zero eigenvalue for L equals the number of connected components (namely, *k*) and the eigenspace is spanned by the indicator vectors of the connected components.

In this case L would form a *block diagonal matrix*:



Spectral Clustering: Graph Laplacian

• Thus, in the "ideal" case for which the data graph consists of k disconnected components, then the multiplicity of the zero eigenvalue for L equals the number of connected components (namely, k) and the eigenspace is spanned by the indicator vectors of the connected components.

• In reality, we don't expect a graph derived from a real similarity matrix to have isolated connected components – that would be too easy ("real data is messy").

Spectral Clustering: Graph Laplacian

• Thus, in the "ideal" case for which the data graph consists of *k* disconnected components, then the multiplicity of the zero eigenvalue for L equals the number of connected components (namely, *k*) and the eigenspace is spanned by the indicator vectors of the connected components.

• In reality, we don't expect a graph derived from a real similarity matrix to have isolated connected components – that would be too easy.

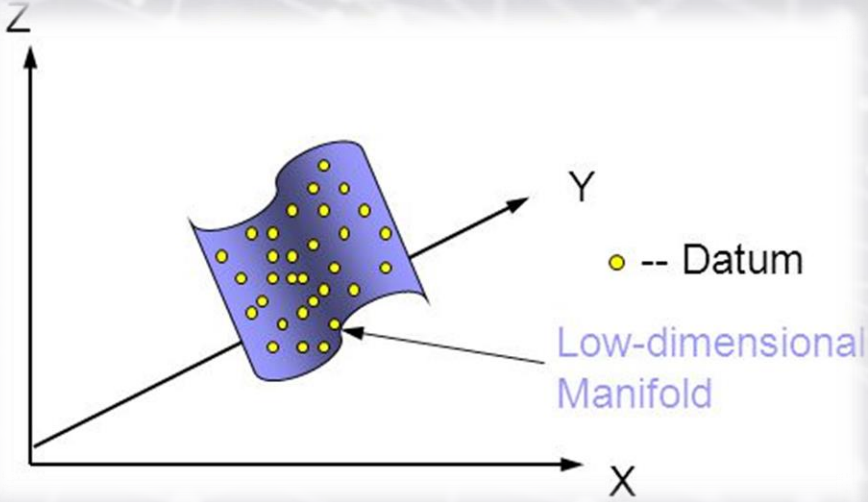
• But it is reasonable to suppose the graph is some small "perturbation" from such a situation. In this case, one can use results from **perturbation theory** to show the eigenvectors of the perturbed Laplacian will be close to these ideal indicator functions.

(*) Briefly stated: the perturbation argument says that if we do not have a completely ideal situation where the between-cluster similarity is exactly 0, but if we have a situation where the between-cluster similarities are very small, then the eigenvectors of the first *k* eigenvalues should be very close to the ones in the ideal case. Thus we should still be able to recover the clustering from those eigenvectors.

Spectral Clustering: Graph Laplacian

• This suggests the following algorithm:

Compute the first *k* eigenvectors of L (k will represent the dimension of our low-dimensional manifold upon which we project the data). Form an n x k (n is the original data set size) matrix U with the eigenvectors of L as the columns. Now one can apply k-means over the rows of U to recover the connected components of the graph. This clustering determines the low-dimensional clustering assignment.



Spectral Clustering: Graph Laplacian

• This suggests the following algorithm:

Compute the first *k* eigenvectors of L (*k* will represent the dimension of the low-dimensional manifold upon which we project the data). Form an n x k (*n* is the original data set size) matrix U with the eigenvectors of L as the columns. Now one can apply k-means over the rows of U to recover the connected components of the graph. This clustering determines the low-dimensional clustering assignment.

(*) In practice, it is important to normalize the graph Laplacian, to account for the fact that some nodes are more highly connected than others. There are two common ways to do with (one using the notion of random walks) – here is the symmetric normalized Laplacian (as used by Ng *et al.*)

$$L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$$

recall that D is the diagonal matrix of vertex degrees.

Spectral Clustering: Graph Laplacian

- This suggests the following algorithm:

Compute the first *k* eigenvectors of L (*k* will represent the dimension of the low-dimensional manifold upon which we project the data). Form an n x k (*n* is the original data set size) matrix U with the eigenvectors of L as the columns. Now one can apply k-means over the rows of U to recover the connected components of the graph. This clustering determines the low-dimensional clustering assignment.

(*) In practice, it is important to normalize the graph Laplacian, to account for the fact that some nodes are more highly connected than others. There are two common ways to do with (one using the notion of random walks) – here is the symmetric normalized Laplacian (as used by Ng *et al.*)

$$L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$$

(*) One can show that this algorithm corresponds with finding a normalized minimum cut in the similarity graph for the data (as discussed on previous slides).

More commonly, researchers use the **normalized cut criterion**, defined as:

$$Ncut(A_1,\ldots,A_K) = \frac{1}{2}\sum_{k=1}^{K}\frac{W(A_k,\bar{A}_k)}{vol(A_k)}$$

where $vol(A_k)=\sum_{i\in A} d_i$, where d_i is the weighted degree of vertex *i* in the similarity graph.

(*) This splits the graph into *K* clusters such that nodes within each cluster are similar to one another, but are different to node in other clusters.

Spectral Clustering: Graph Laplacian

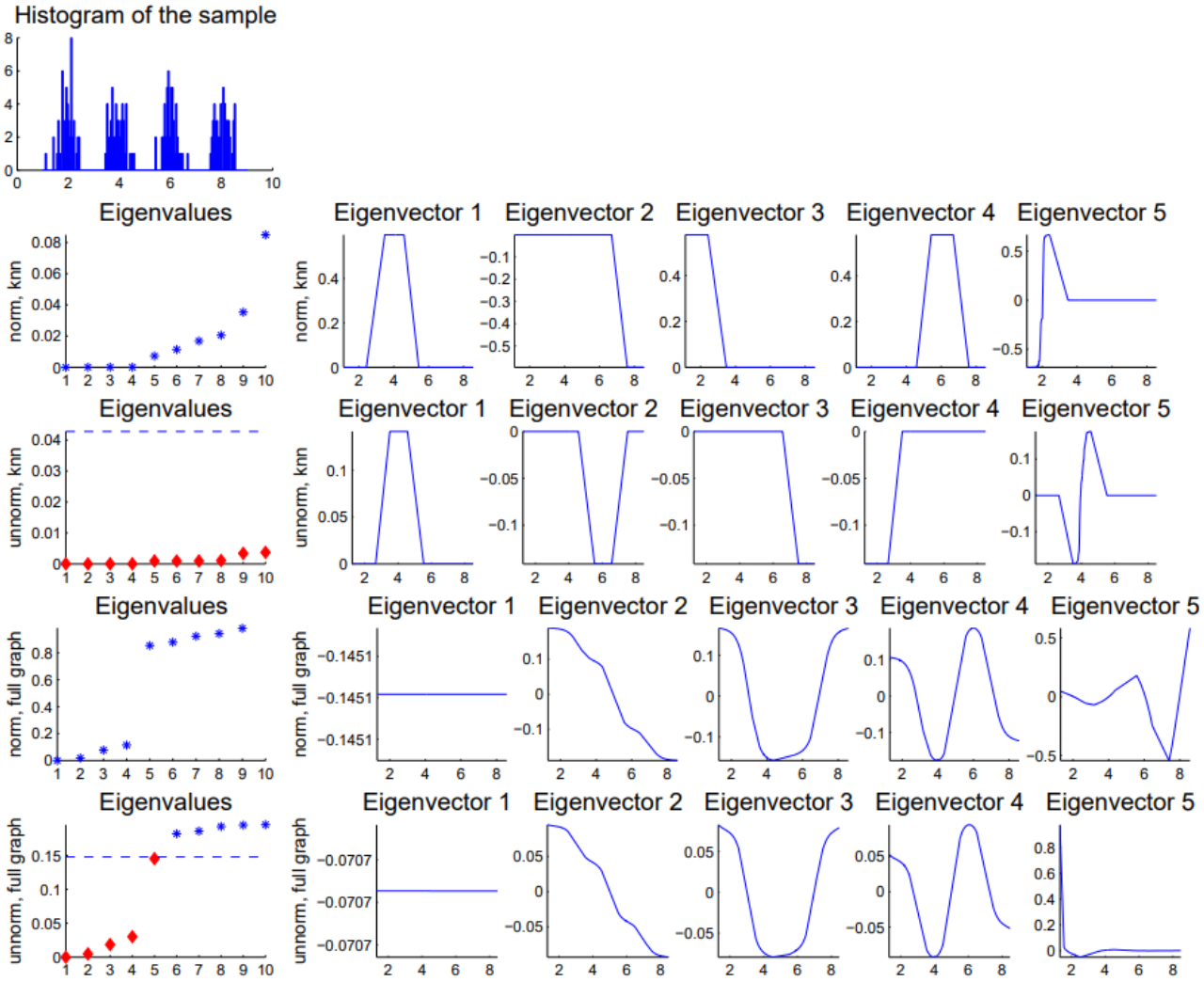


Figure 1: Toy example for spectral clustering. Left upper corner: histogram of the data. First and second row: eigenvalues and eigenvectors of L_{rw} and L based on the k-nearest neighbor graph. Third and fourth row: eigenvalues and eigenvectors of L_{rw} and L based on the fully connected graph. For all plots, we used we use the Gaussian kernel with $\sigma = 1$ as similarity function.

Spectral Clustering

On Spectral Clustering: Analysis and an algorithm

Andrew Y. Ng
CS Division
U.C. Berkeley
ang@cs.berkeley.edu

Michael I. Jordan
CS Div. & Dept. of Stat.
U.C. Berkeley
jordan@cs.berkeley.edu

Yair Weiss
School of CS & Engr.
The Hebrew Univ.
yweiss@cs.huji.ac.il

Abstract

Despite many empirical successes of *spectral clustering* methods—algorithms that cluster points using eigenvectors of matrices derived from the data—there are several unresolved issues. First, there are a wide variety of algorithms that use the eigenvectors in slightly different ways. Second, many of these algorithms have no proof that they will actually compute a reasonable clustering. In this paper, we present a simple spectral clustering algorithm that can be implemented using a few lines of Matlab. Using tools from matrix perturbation theory, we analyze the algorithm, and give conditions under which it can be expected to do well. We also show surprisingly good experimental results on a number of challenging clustering problems.

(*) NIPS 2001 paper, Ng et al., on spectral clustering ~6500 citations

Spectral Clustering

On Spectral Clustering: Analysis and an algorithm

Andrew Y. Ng
CS Division
U.C. Berkeley
ang@cs.berkeley.edu

Michael I. Jordan
CS Div. & Dept. of Stat.
U.C. Berkeley
jordan@cs.berkeley.edu

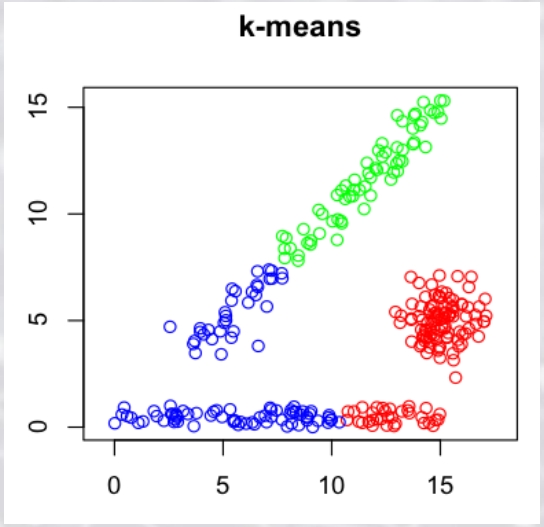
Yair Weiss
School of CS & Engr.
The Hebrew Univ.
yweiss@cs.huji.ac.il

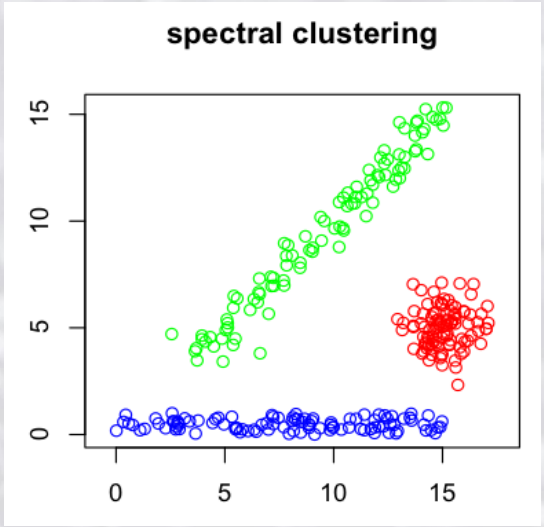
2 Algorithm

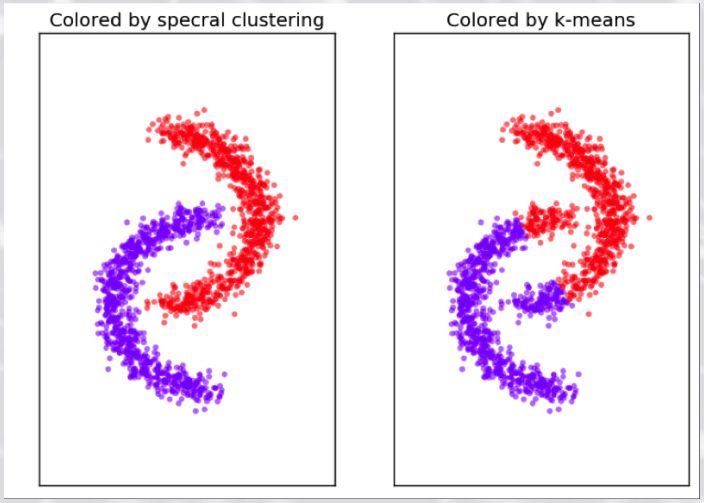
Given a set of points $S = \{s_1, \ldots, s_n\}$ in $\mathbb{R}^l$ that we want to cluster into k subsets:

1. Form the affinity matrix $A \in \mathbb{R}^{n \times n}$ defined by $A_{ij} = \exp(-||s_i - s_j||^2/2\sigma^2)$ if $i \neq j$, and $A_{ii} = 0$.
2. Define D to be the diagonal matrix whose (i,i)-element is the sum of A's i-th row, and construct the matrix $L = D^{-1/2}AD^{-1/2}$.[1]
3. Find $x_1, x_2, \ldots, x_k$, the k largest eigenvectors of L (chosen to be orthogonal to each other in the case of repeated eigenvalues), and form the matrix $X = [x_1 x_2 \ldots x_k] \in \mathbb{R}^{n \times k}$ by stacking the eigenvectors in columns.
4. Form the matrix Y from X by renormalizing each of X's rows to have unit length (i.e. $Y_{ij} = X_{ij}/(\sum_j X_{ij}^2)^{1/2}$).
5. Treating each row of Y as a point in $\mathbb{R}^k$, cluster them into k clusters via K-means or any other algorithm (that attempts to minimize distortion).
6. Finally, assign the original point s_i to cluster j if and only if row i of the matrix Y was assigned to cluster j.

Spectral Clustering





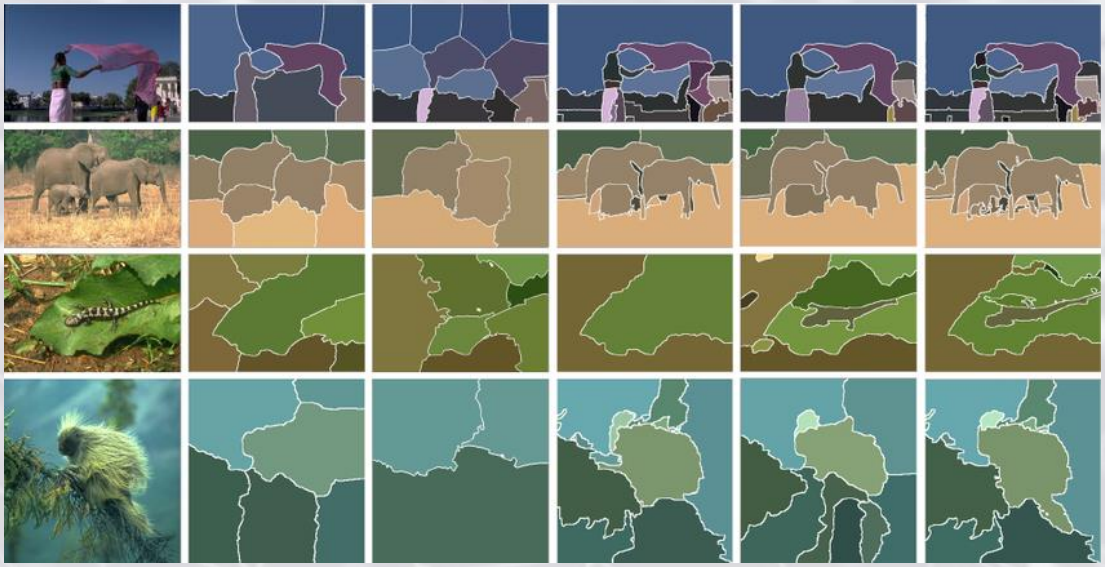


Tutorial on spectral clustering: https://arxiv.org/pdf/0711.0189.pdf

Github demo: https://github.com/pin3da/spectral-clustering (python)

Spectral Clustering: Applications

Naturally, one can use spectral clustering for image segmentation applications (in addition to various dimensionality reduction applications).



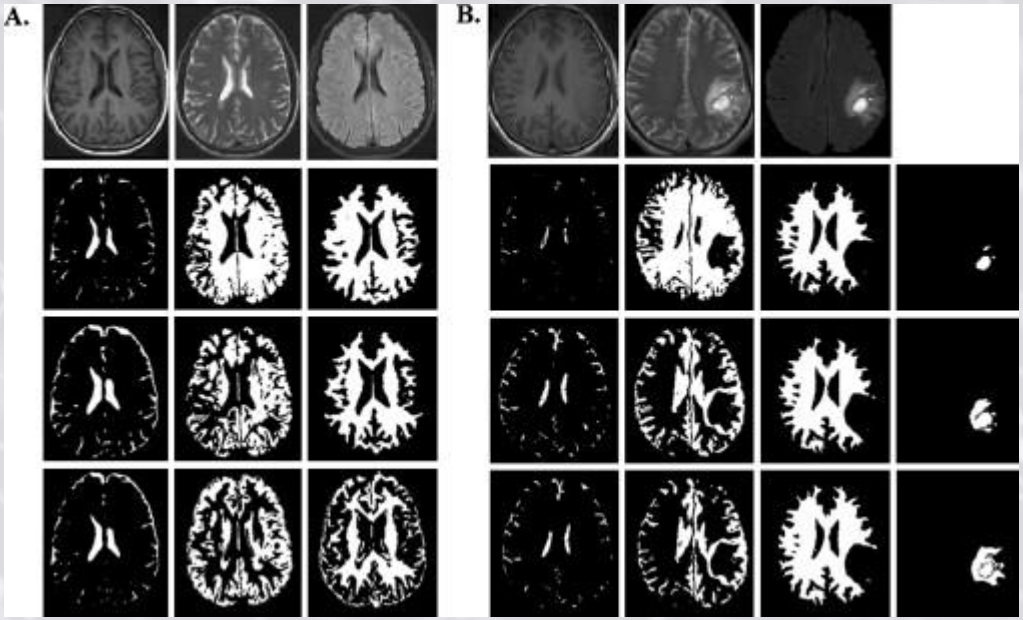
Segmentation based on spectral clustering.

(*) A significant challenge for image segmentation applications of spectral clustering relates to the construction/definition of the Laplacian (i.e. similarity/affinity) matrix. This is a deep question that concerns feature selection more broadly in ML and data mining.

Spectral Clustering: Applications

Sindhumol, et al. "Spectral clustering independent component analysis for tissue classification from brain MRI"

The authors use a variant of spectral clustering (*spectral clustering ICA*) to improve brain tissue classification from MRI scans; results were 98% accuracy for clinical abnormality analysis.



https://www.sciencedirect.com/science/article/pii/S174680941300092X

ISOMAP

- Like spectral clustering, Isomap is a non-linear dimensionality reduction method.

- More specifically, Isomap is an **isometric mapping** method (isometric mappings preserve distance) that provides an extension of a general class of algorithms known as **metric multidimensional scaling** (MDS) methods.

ISOMAP

• Like spectral clustering, Isomap is a non-linear dimensionality reduction method.

• More specifically, Isomap is an **isometric mapping** method (isometric mappings preserve distance) that provides an extension of a general class of algorithms known as metric multidimensional scaling (MDS) methods.

• **MDS** performs a low-dimensional embedding of a data set based on pairwise distances between data points (simply by using straight-line Euclidean distance). Alternatively, Isomap incorporates instead the **geodesic distance** between two vertices in a graph.

ISOMAP

• Like spectral clustering, Isomap is a non-linear dimensionality reduction method.

• More specifically, Isomap is an **isometric mapping** method (isometric mappings preserve distance) that provides an extension of a general class of algorithms known as metric multidimensional scaling (MDS) methods.

• **MDS** performs a low-dimensional embedding of a data set based on pairwise distances between data points (simply by using straight-line Euclidean distance). Alternatively, Isomap incorporates instead the **geodesic distance** between two vertices in a graph.

• The geodesic distance is defined as the sum of the edge weights in a shortest path between two vertices.

Q: How is geodesic distance more informative than Euclidean distance?

ISOMAP

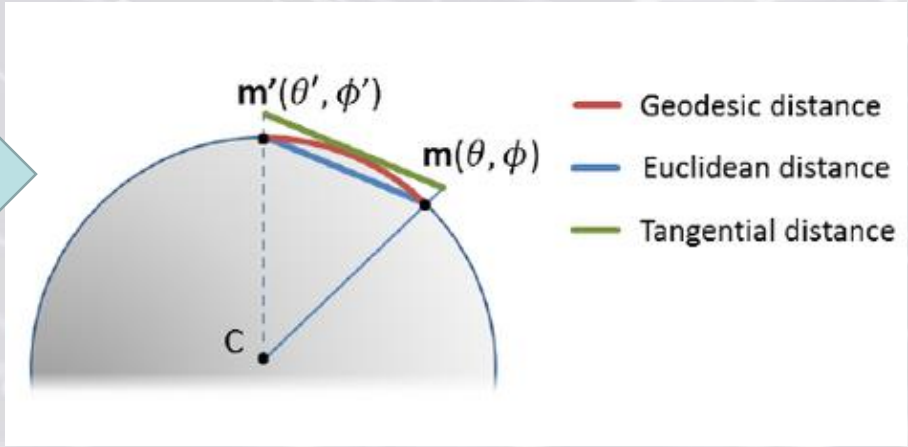
• Like spectral clustering, Isomap is a non-linear dimensionality reduction method.

• More specifically, Isomap is an **isometric mapping** method (isometric mappings preserve distance) that provides an extension of the metric multidimensional scaling (MDS) method.

• MDS performs a low-dimensional embedding of a data set based on pairwise distances between data points (simply by using straight-line Euclidean distance). Alternatively, Isomap incorporates instead the **geodesic distance** between two vertices in a graph.

• The geodesic distance is defined as the sum of the edge weights in a shortest path between two vertices.

Geodesic distance incorporates the manifold structure in the resulting embedding.



ISOMAP

A Global Geometric Framework for Nonlinear Dimensionality Reduction

Joshua B. Tenenbaum,[1]* Vin de Silva,[2] John C. Langford[3]

Scientists working with large volumes of high-dimensional data, such as global climate patterns, stellar spectra, or human gene distributions, regularly confront the problem of dimensionality reduction: finding meaningful low-dimensional structures hidden in their high-dimensional observations. The human brain confronts the same problem in everyday perception, extracting from its high-dimensional sensory inputs—30,000 auditory nerve fibers or 10^6 optic nerve fibers—a manageably small number of perceptually relevant features. Here we describe an approach to solving dimensionality reduction problems that uses easily measured local metric information to learn the underlying global geometry of a data set. Unlike classical techniques such as principal component analysis (PCA) and multidimensional scaling (MDS), our approach is capable of discovering the nonlinear degrees of freedom that underlie complex natural observations, such as human handwriting or images of a face under different viewing conditions. In contrast to previous algorithms for nonlinear dimensionality reduction, ours efficiently computes a globally optimal solution, and, for an important class of data manifolds, is guaranteed to converge asymptotically to the true structure.

A canonical problem in dimensionality re- ality may be quite high (e.g., 4096 for these

The original Isomap paper, by Tenebaum *et al*. (same Tenebaum from Bayesian concept learning as discussed previously, Science, December 2000, ~11k citations.

http://web.mit.edu/cocosci/Papers/sci_reprint.pdf

ISOMAP

• The Isomap algorithm combines the major algorithmic features of PCA and MDS – computational efficiency, global optimality, and asymptotic convergence guarantees – with the flexibility to learn a broad class of non-linear manifolds.

• In particular, Isomap seeks to preserve the intrinsic geometry of the data, as captured in the geodesic manifold distances between pairs of points.

ISOMAP

• The Isomap algorithm combines the major algorithmic features of PCA and MDS – computational efficiency, global optimality, and asymptotic convergence guarantees – with the flexibility to learn a broad class of non-linear manifolds.

• In particular, Isomap seeks to preserve the intrinsic geometry of the data, as captured in the geodesic manifold distances between pairs of points.

• The key step deals with estimating the geodesic distance between distant points, given only input-space distances.

• For neighboring points, the input-space distance provides a good approximation to geodesic distance. For faraway points, the geodesic distance can be approximated by adding up a sequence of "short hops" between neighboring points. These approximations can be computed by finding a shortest path.

ISOMAP

(*) The Isomap algorithm is comprised of (3) general steps:

(1) The **first step** determines which points are neighbors on the manifold *M*, based on the distance $d_X(i,j)$ between pairs of points *i*,j in the input space *X*.

As we mentioned previously in the this lecture, there are several options for determining "neighborhoods", including the use of k-NN or to connect a point to each point within some fixed radius ε, etc.

From here, we construct the neighborhood graph G (again, each point, say, could be connected to another if it is a K nearest neighbor, or the edge weights could simply equal the Euclidean distance between points).

ISOMAP

(*) The Isomap algorithm is comprised of (3) general steps:

(2) In its **second step**, Isomap estimates the geodesic distances $d_M(i,j)$ between all pairs of points on the manifold M by computing their shortest path distances $d_G(i,j)$ in the graph G.

Q: What is a well-known algorithm to compute these geodesic distance (i.e. shortest paths in G)?

ISOMAP

(*) The Isomap algorithm is comprised of (3) general steps:

(2) In its second step, Isomap estimates the geodesic distances $d_M(i,j)$ between all pairs of points on the manifold M by computing their shortest path distances $d_G(i,j)$ in the graph G.

Q: What is a well-known algorithm to compute these geodesic distance (i.e. shortest paths in G)?

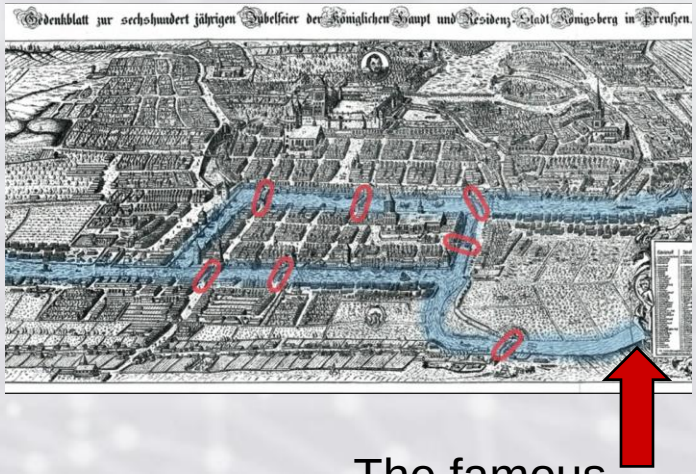
There are several options: **Dijkstra's Algorithm** is perhaps the best-known (one can also can use Floyd-Warshall, for instance).

Aside: Dijkstra's Algorithm

Shortest path network.

- Directed graph G = (V, E).
- Source s, destination t.
- Length ℓ_e = length of edge e.

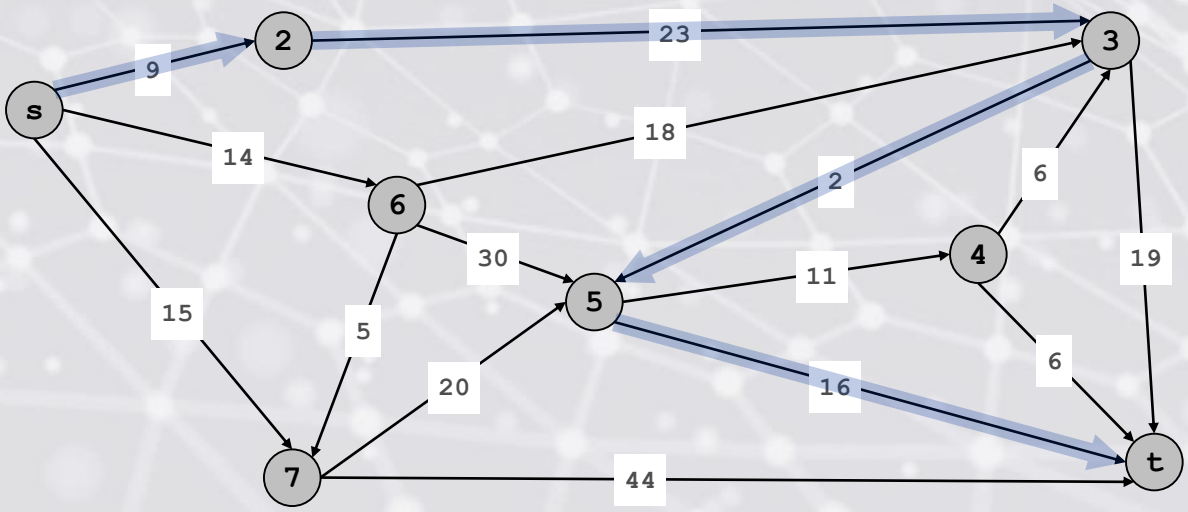
Euler



The famous "seven bridges of Koningsberg" problem that led to the inception of graph theory

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

Aside: Dijkstra's Algorithm

Dijkstra's algorithm.

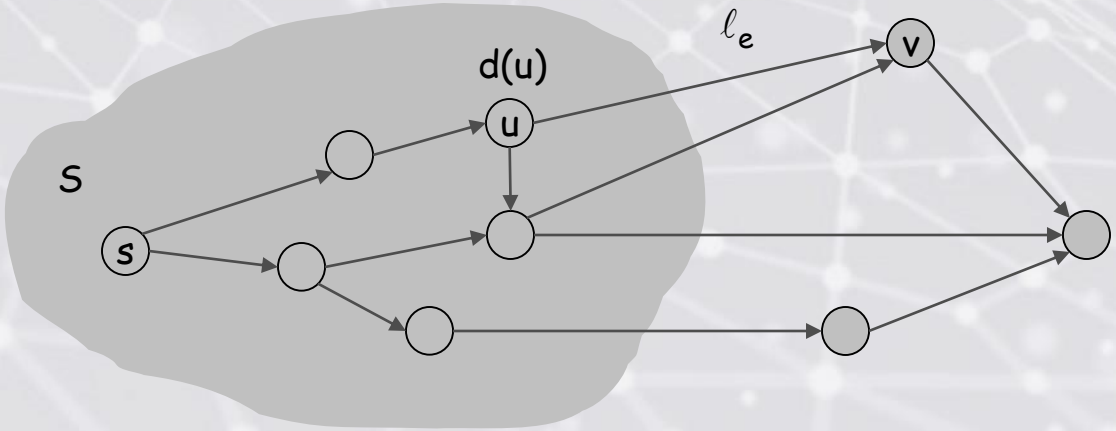
- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize $S = \{ s \}$, $d(s) = 0$.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e ,$$

add v to S, and set $d(v) = \pi(v)$.

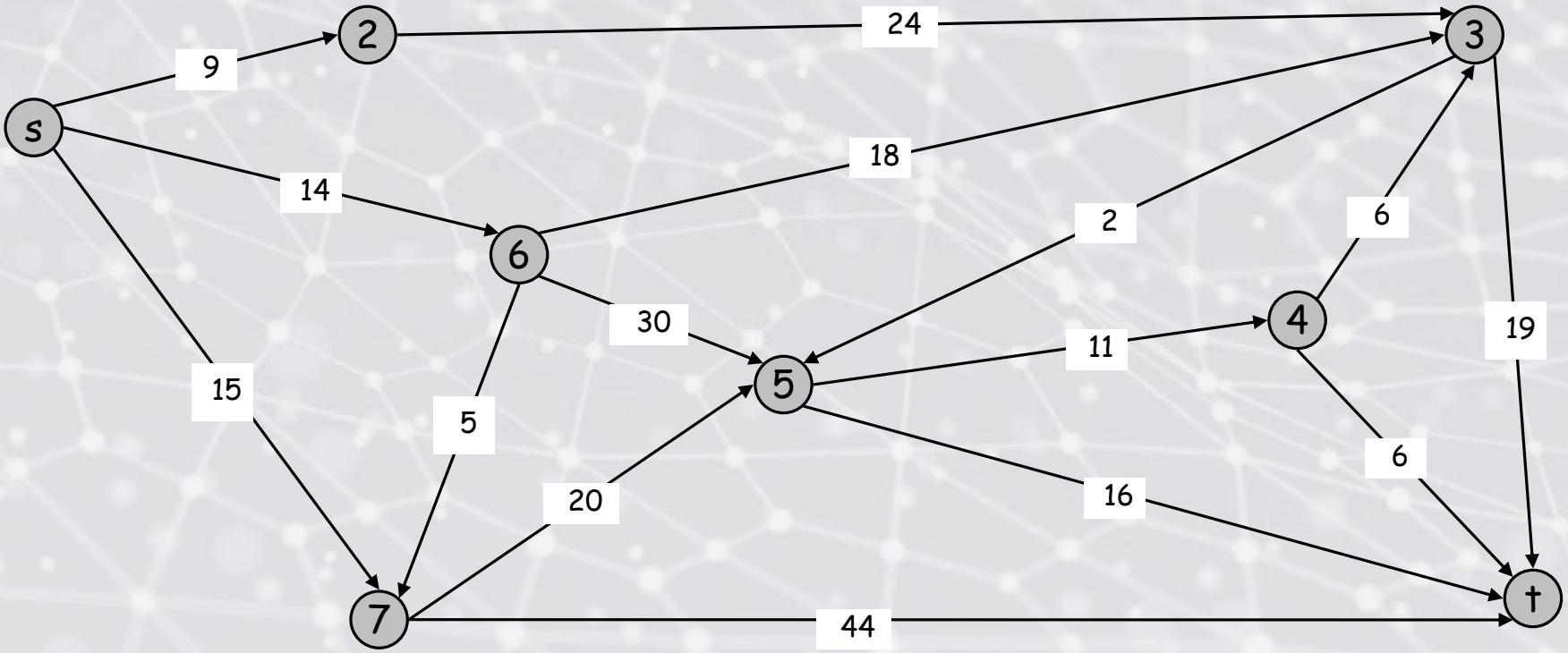
shortest path to some u in explored part, followed by a single edge (u, v)

- (Note: $\pi(v)$ represents a "temporary" distance label the algorithm runs)



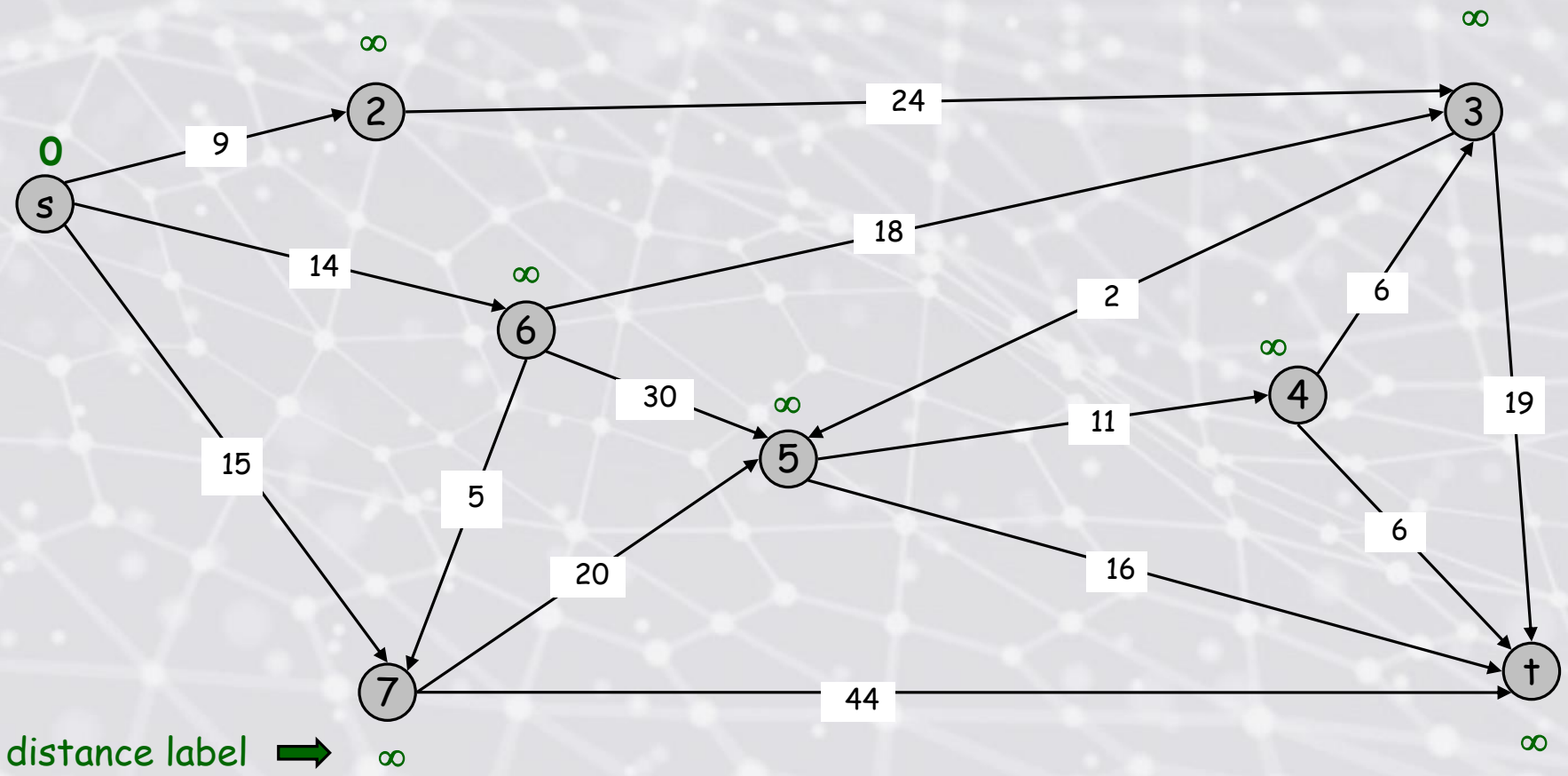
Aside: Dijkstra's Algorithm

Find shortest path from s to t.



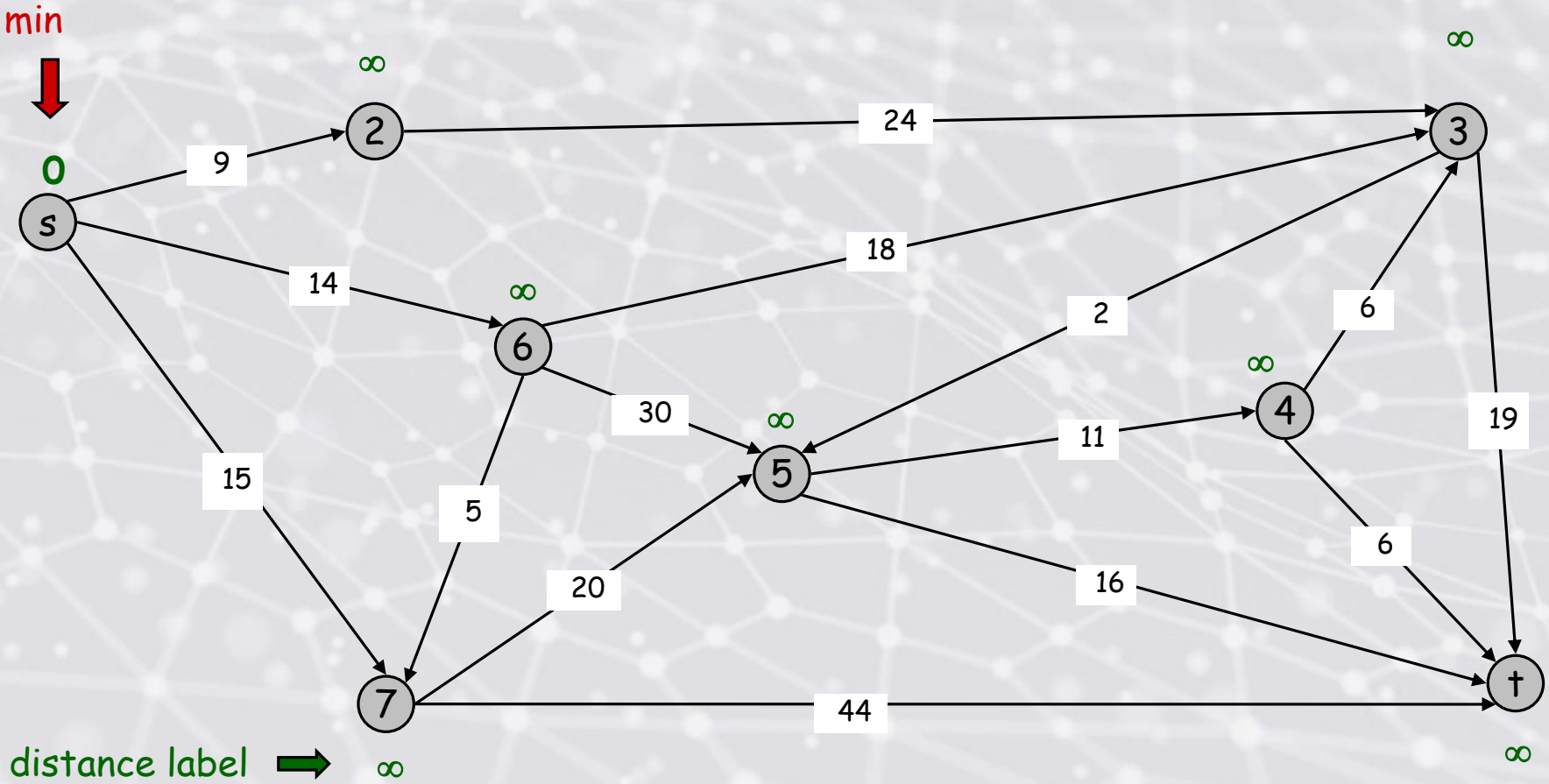
Aside: Dijkstra's Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



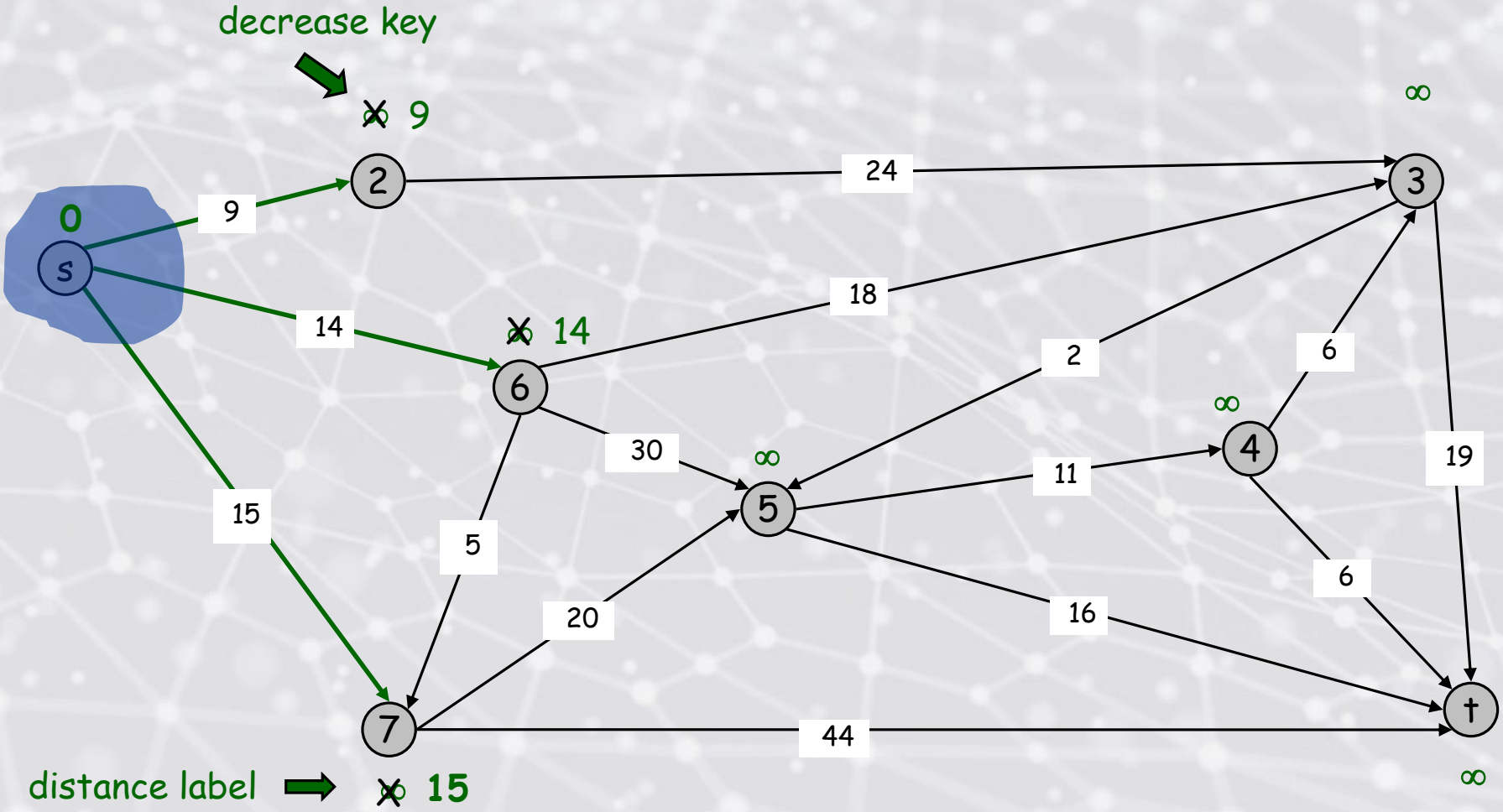
Aside: Dijkstra's Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



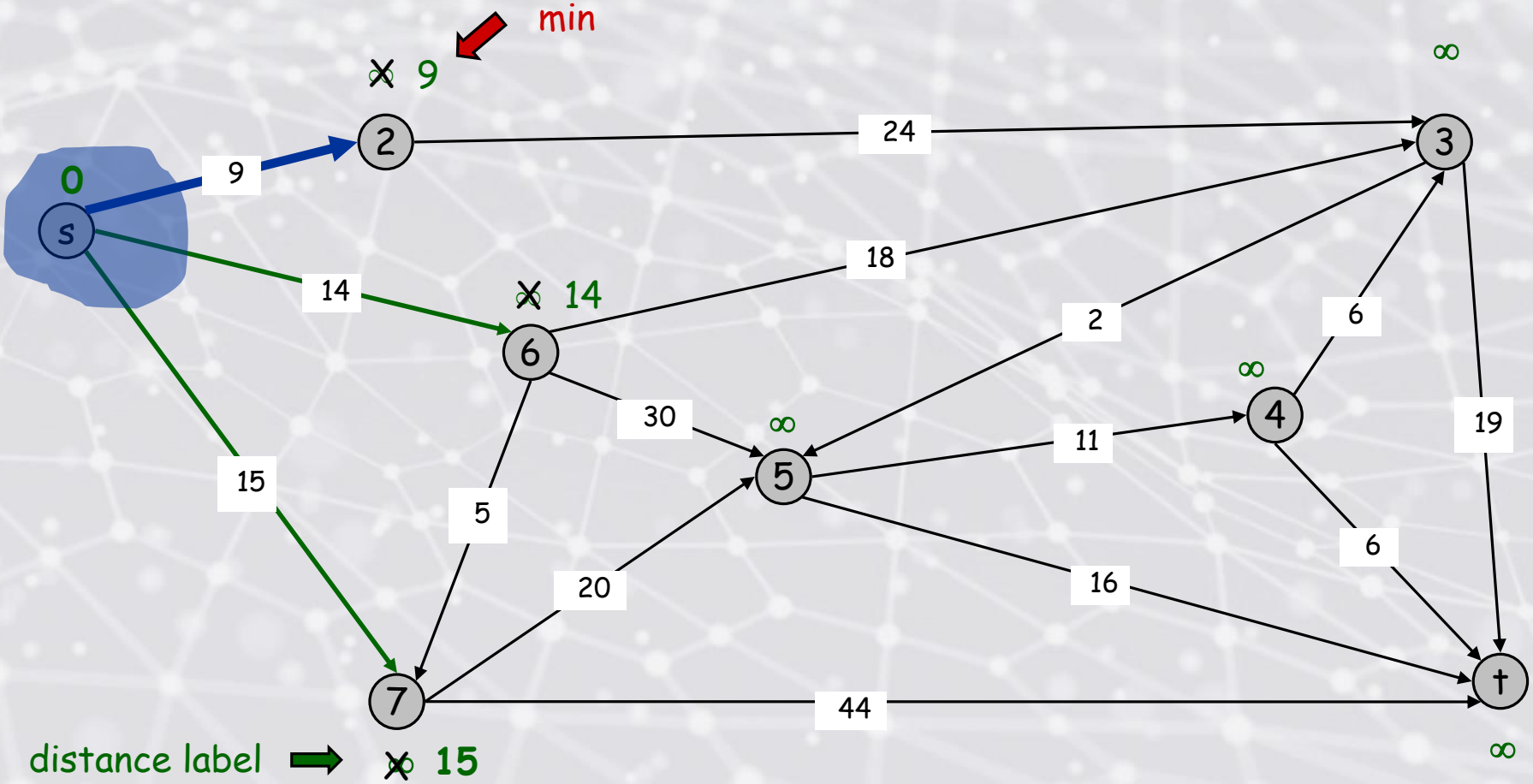
Aside: Dijkstra's Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



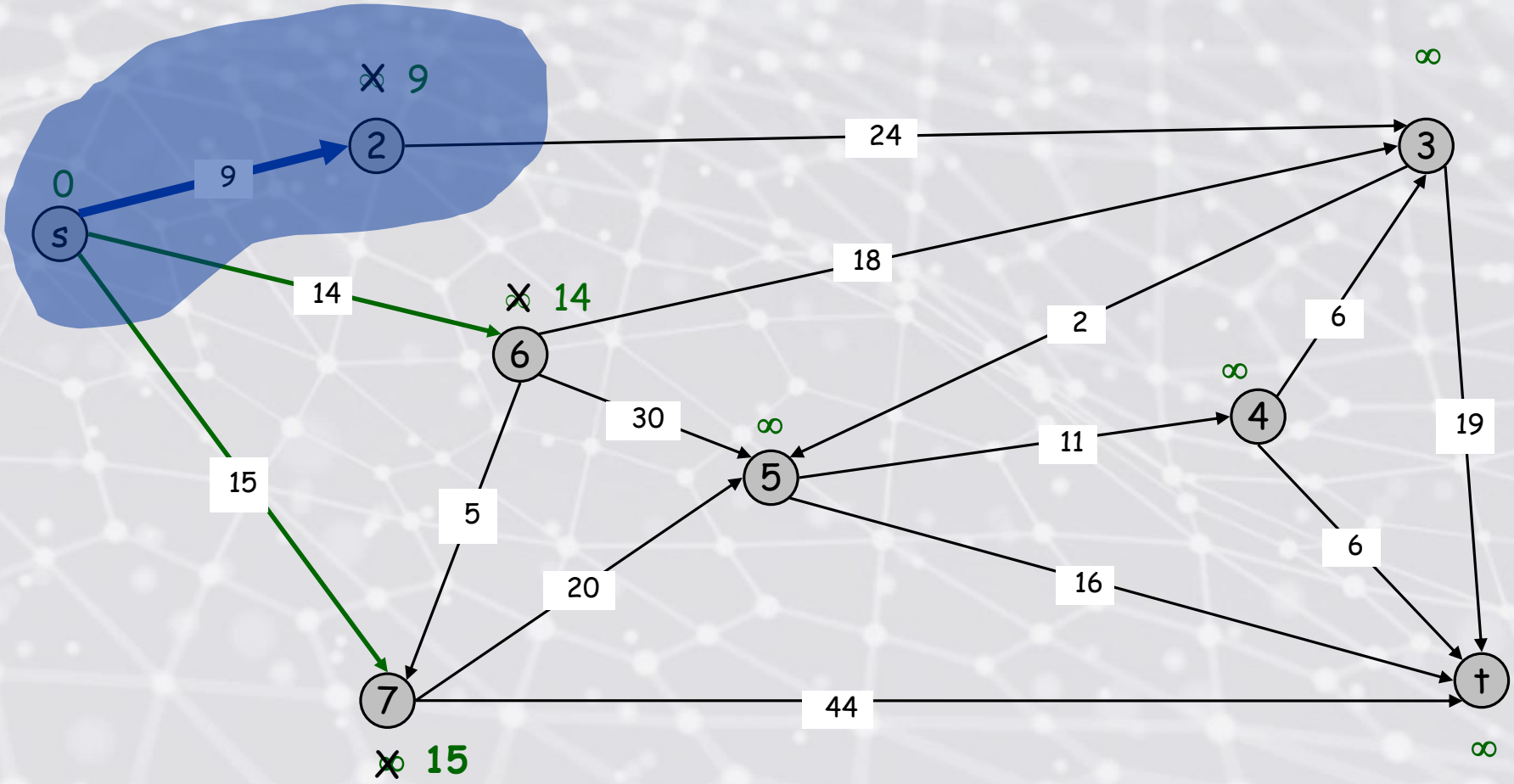
Aside: Dijkstra's Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



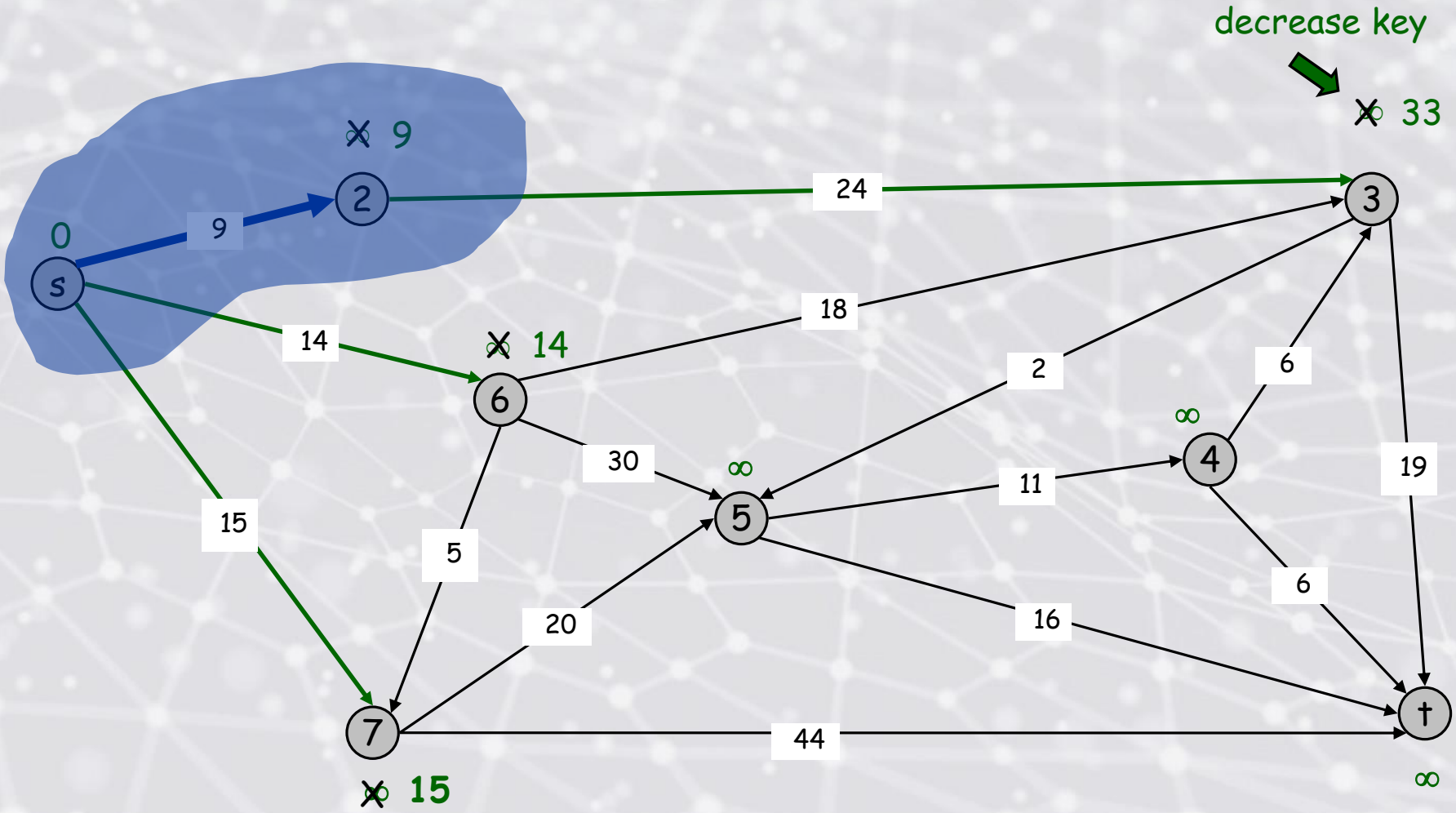
Aside: Dijkstra's Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



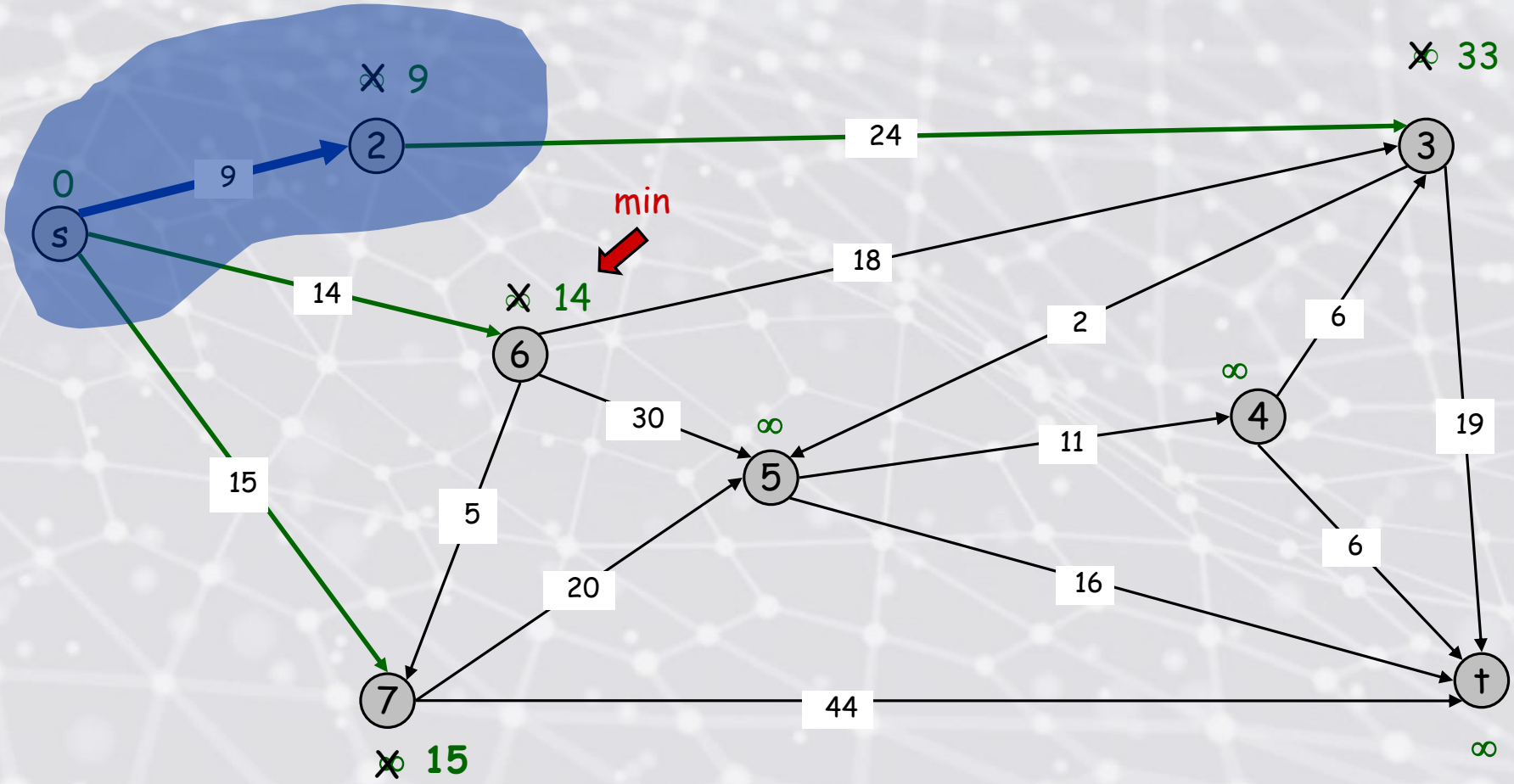
Aside: Dijkstra's Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



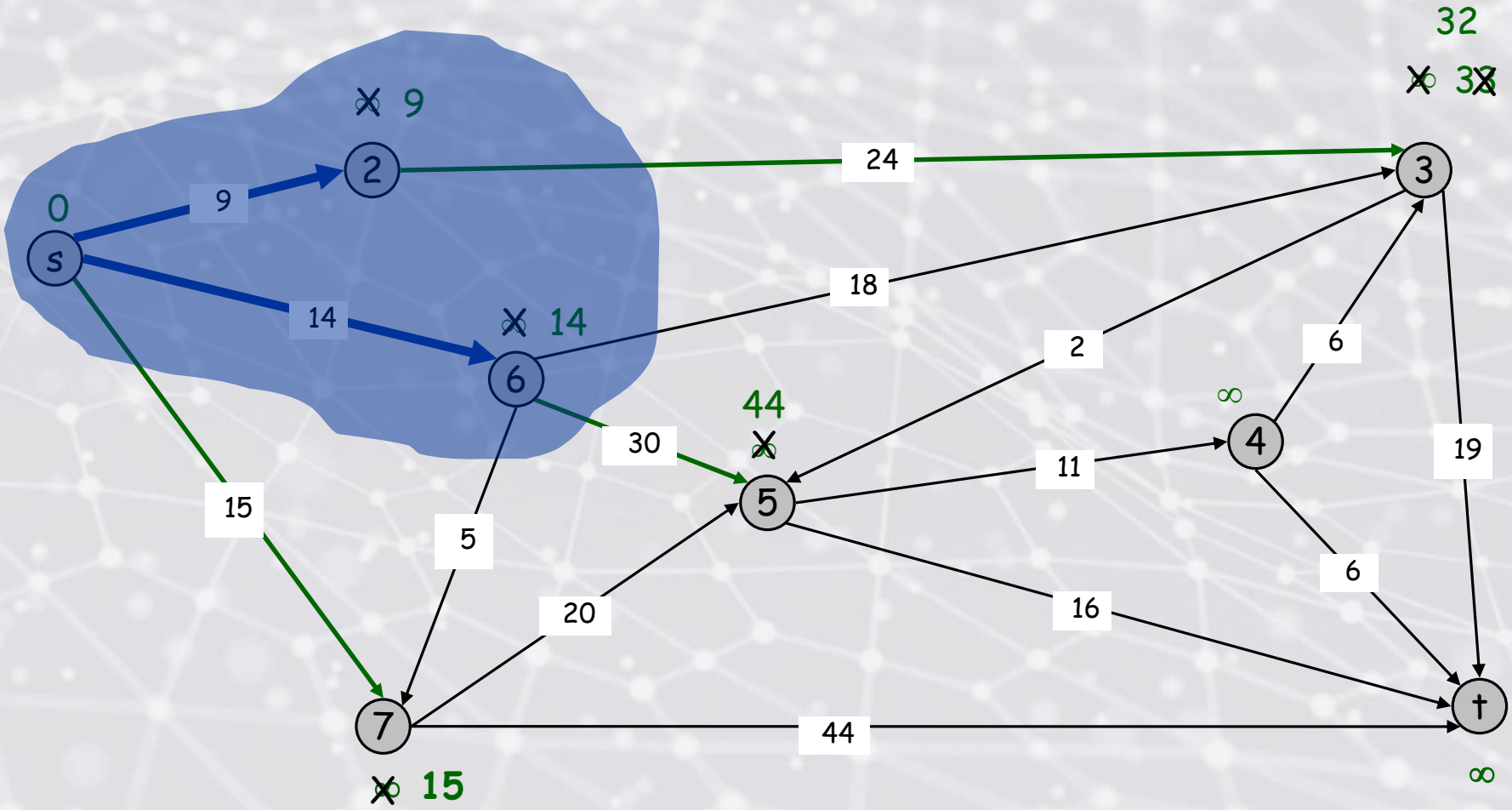
Aside: Dijkstra's Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



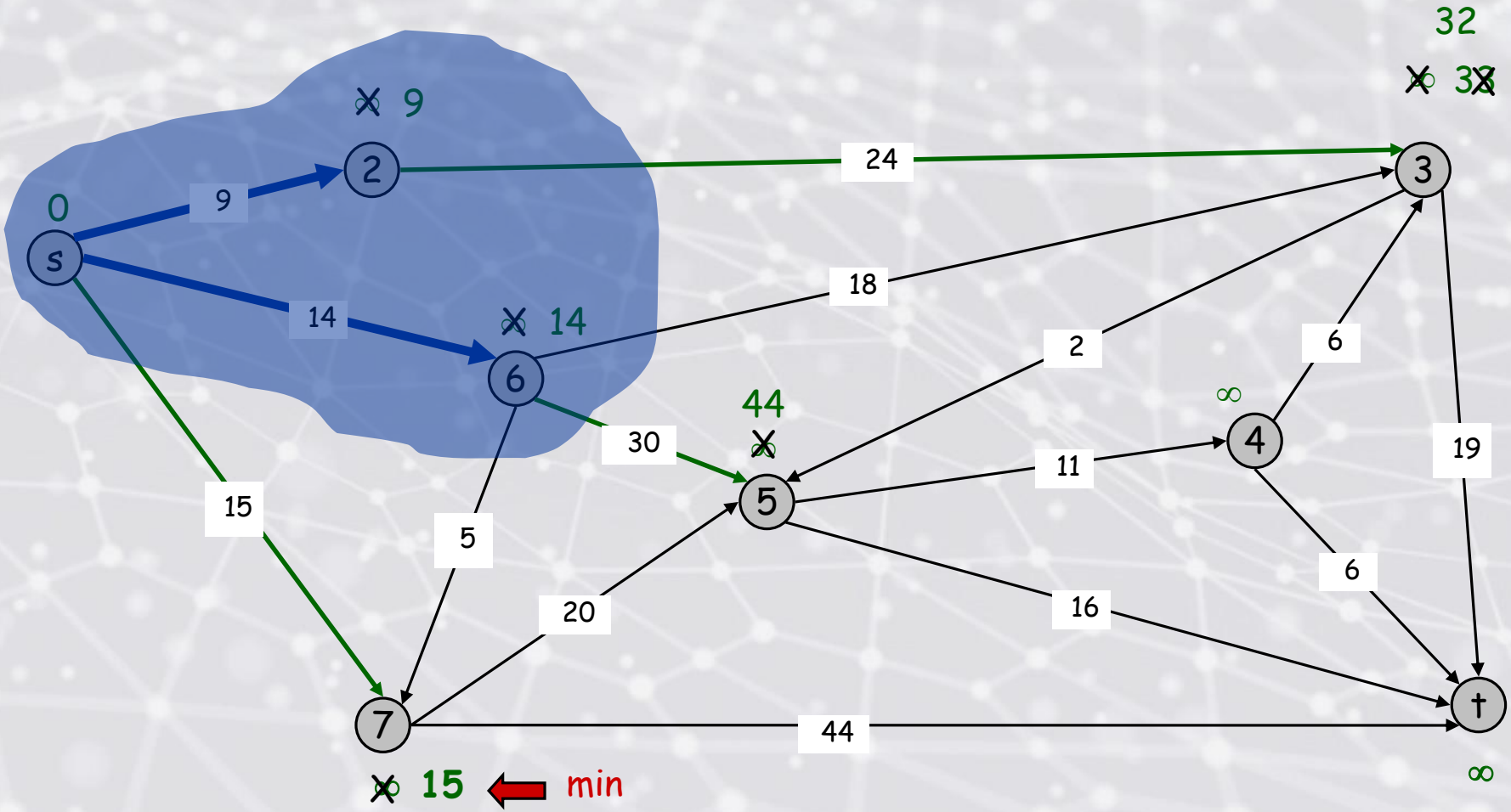
Aside: Dijkstra's Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



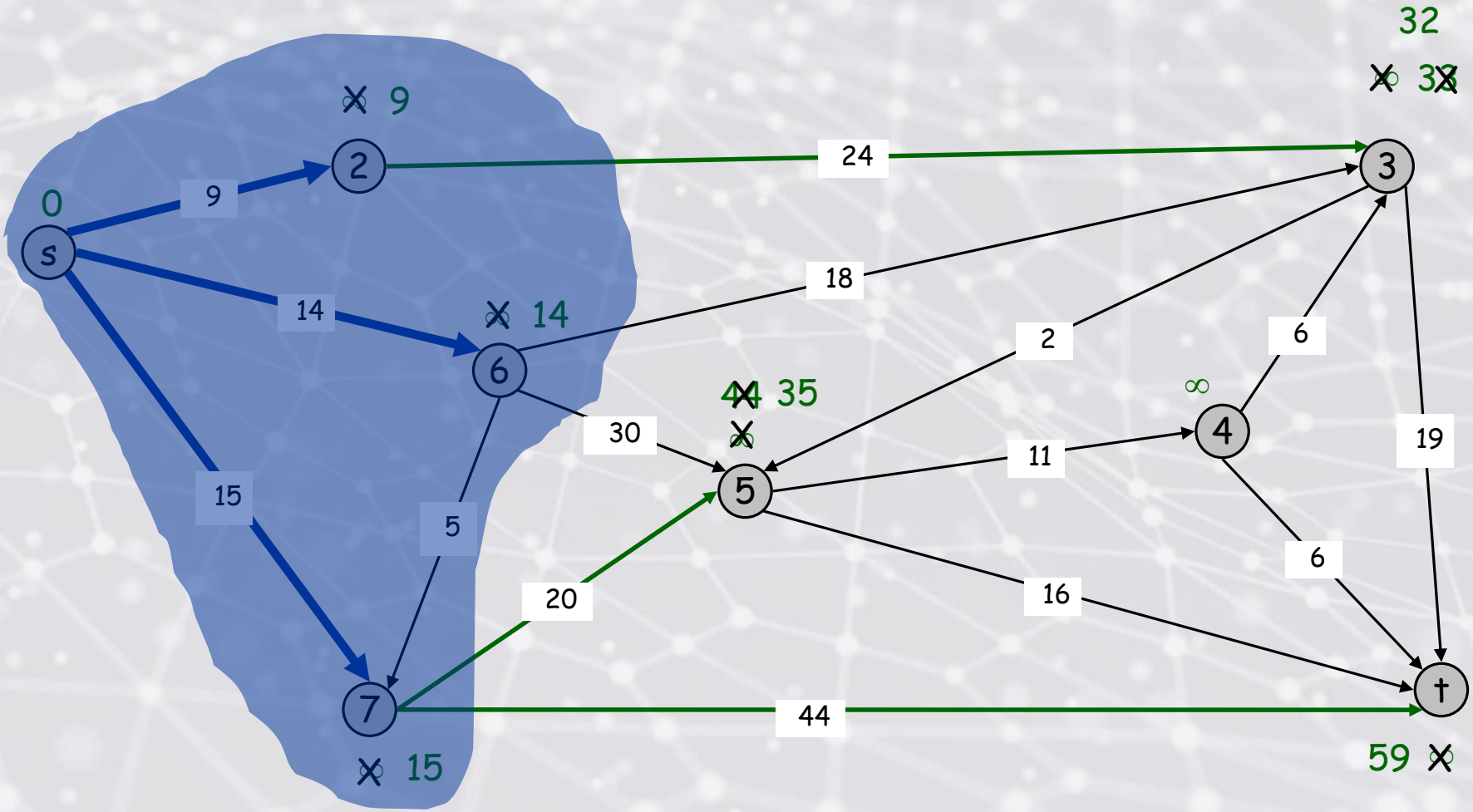
Aside: Dijkstra's Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



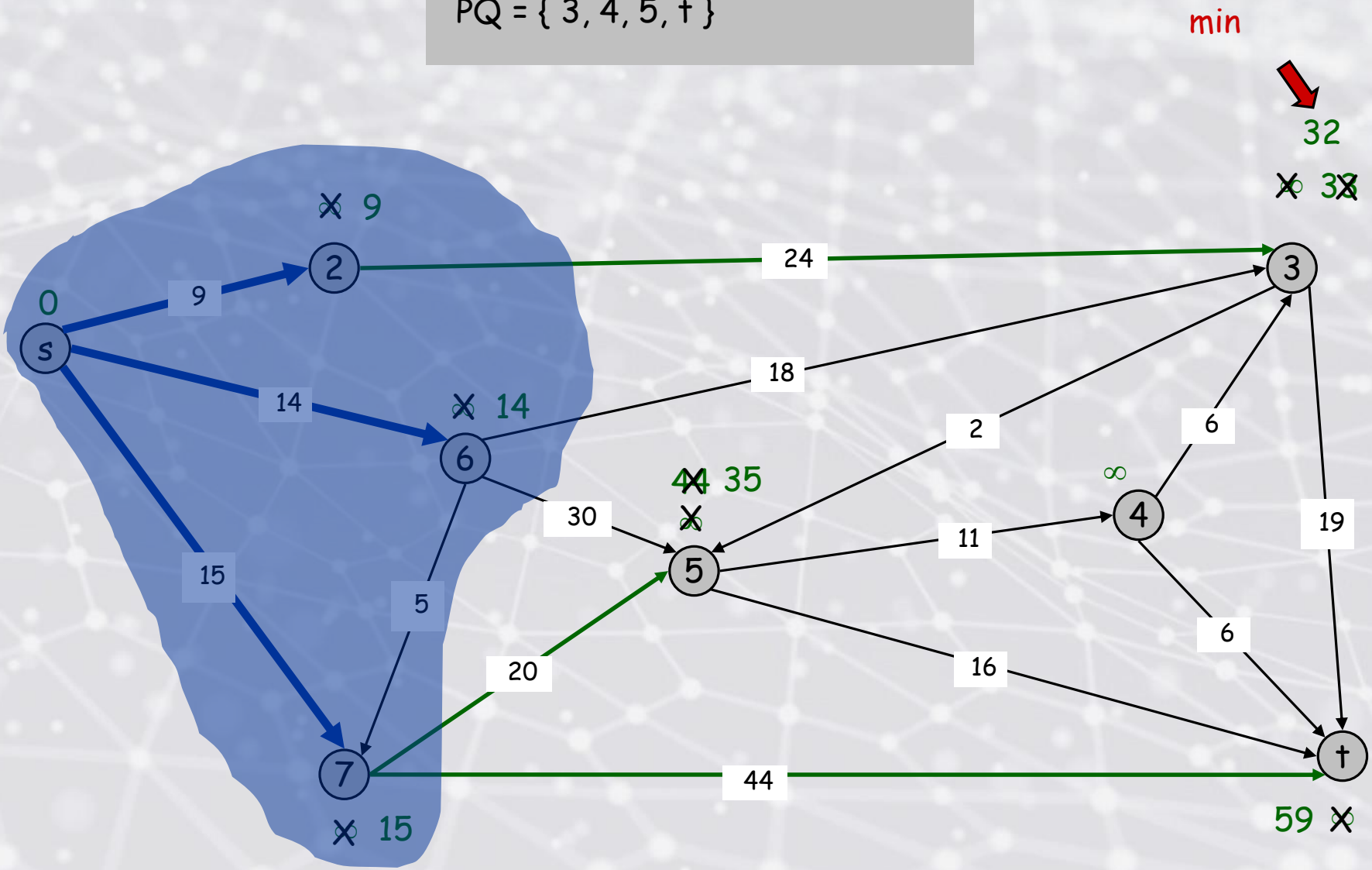
Aside: Dijkstra's Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



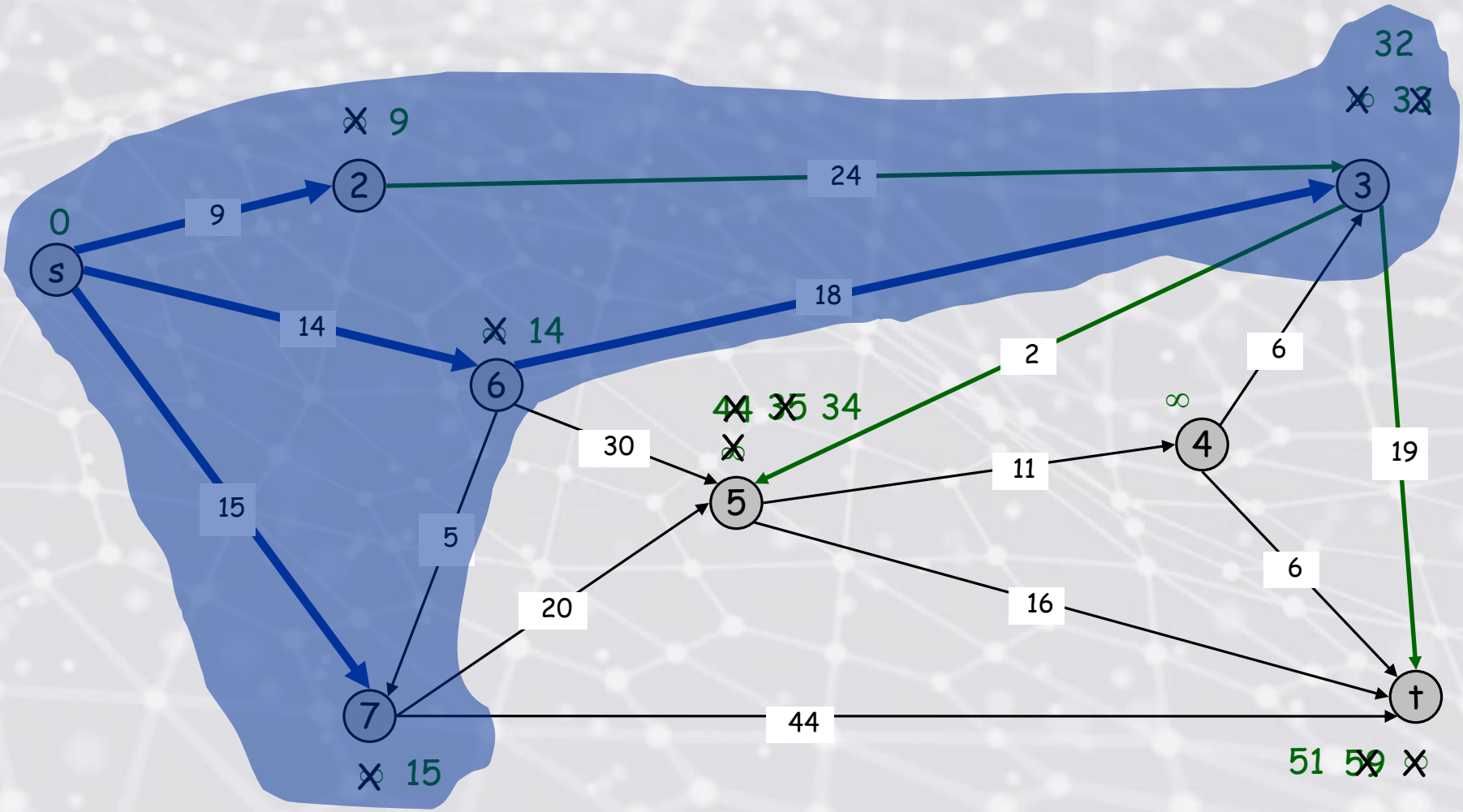
Aside: Dijkstra's Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



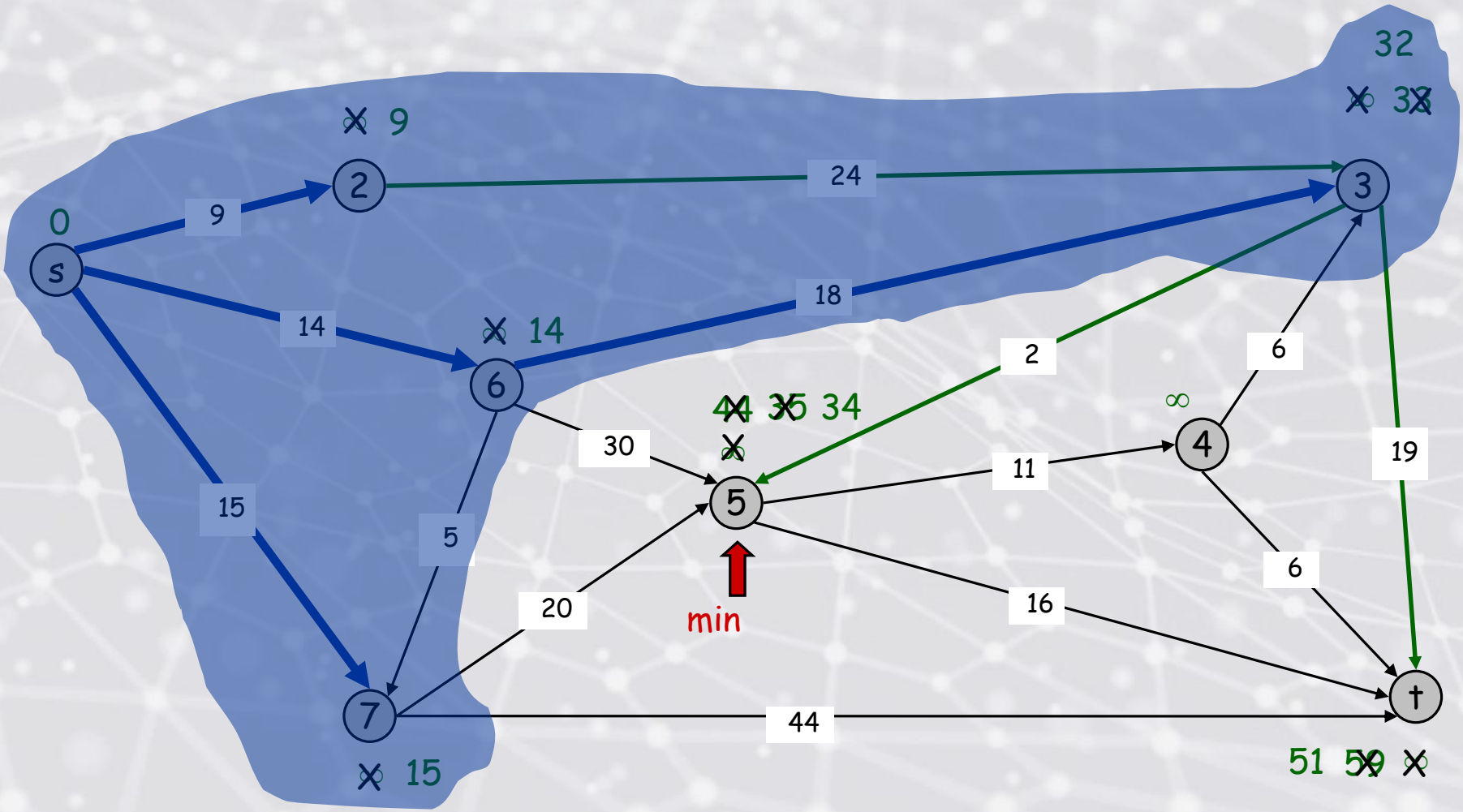
Aside: Dijkstra's Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



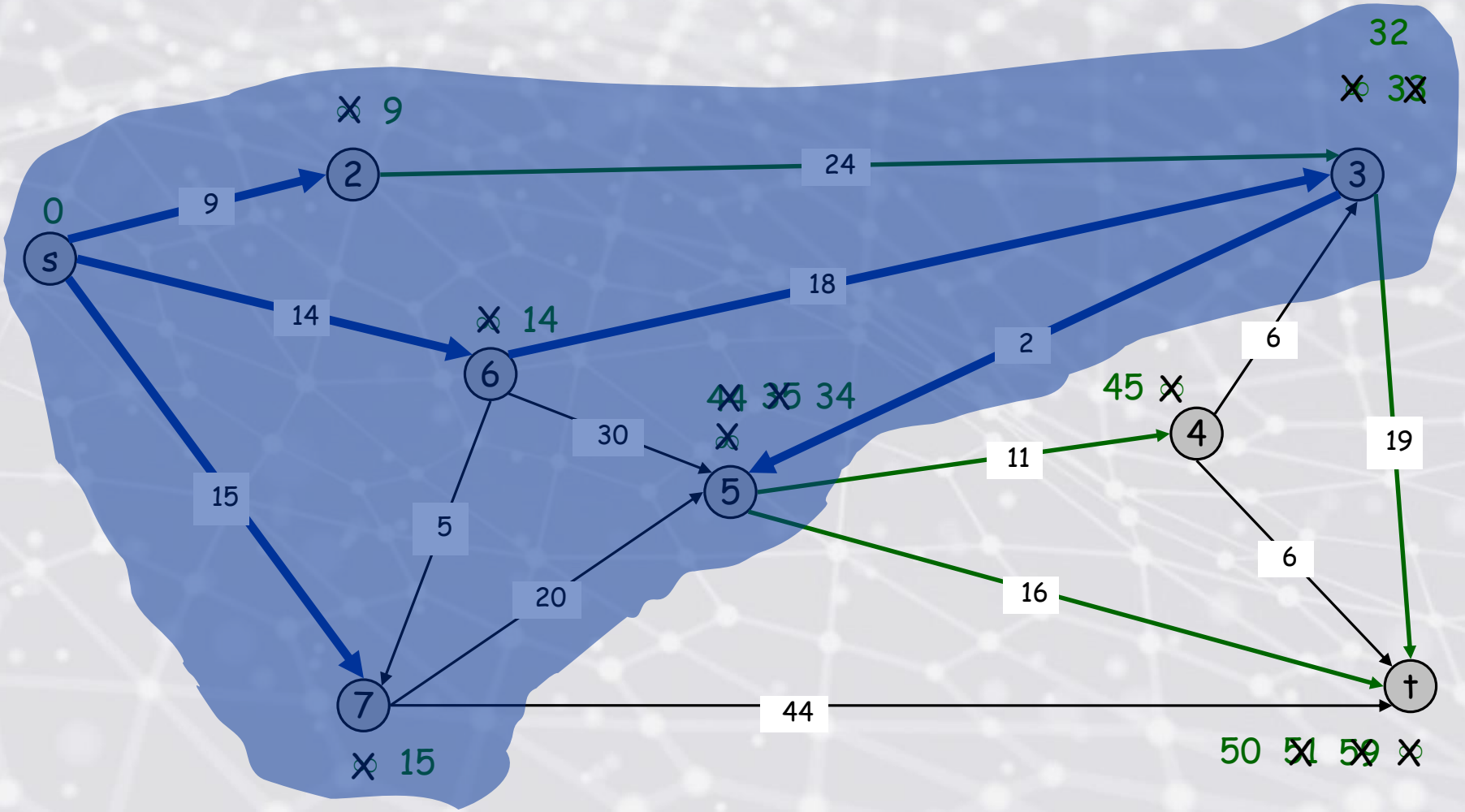
Aside: Dijkstra's Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



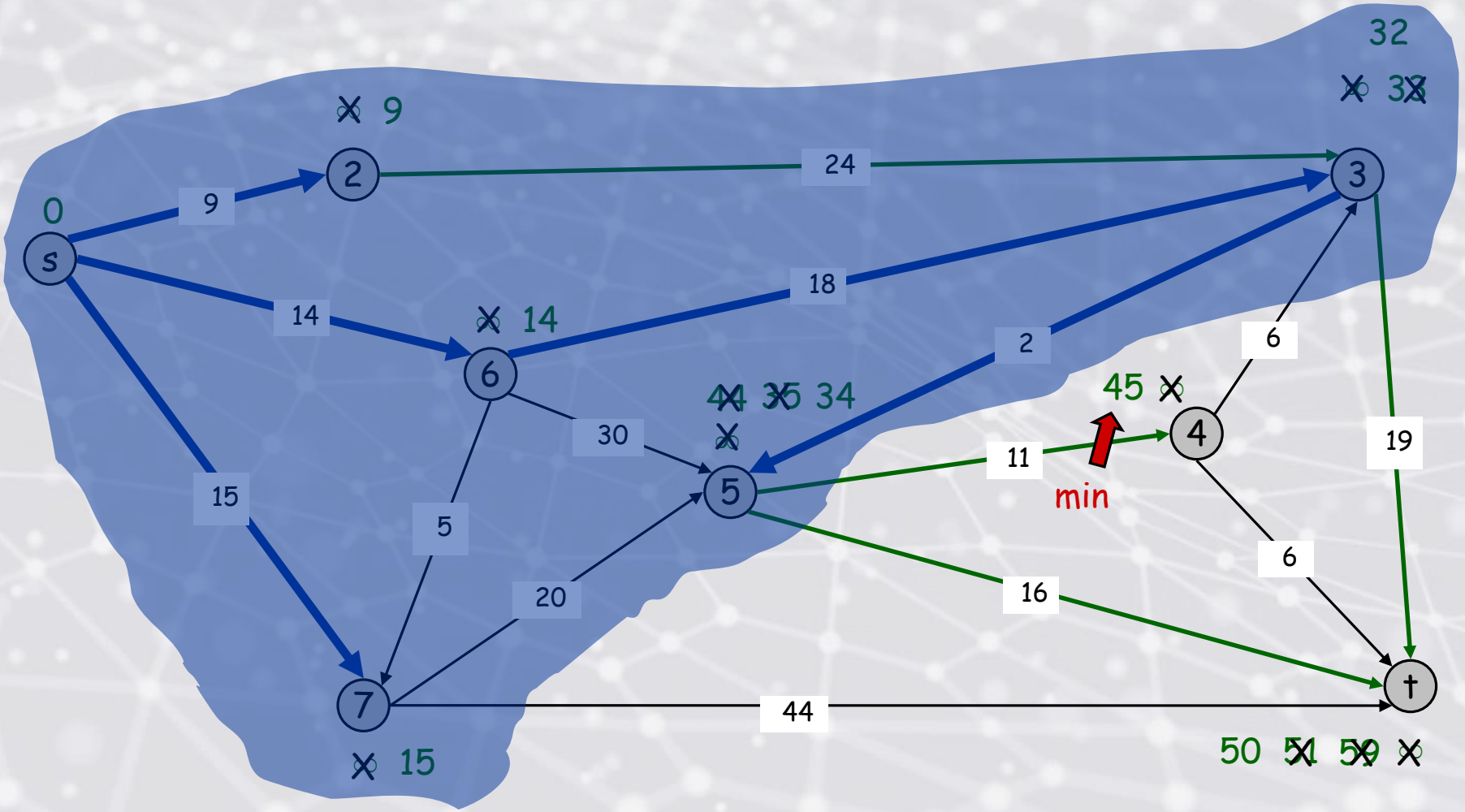
Aside: Dijkstra's Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



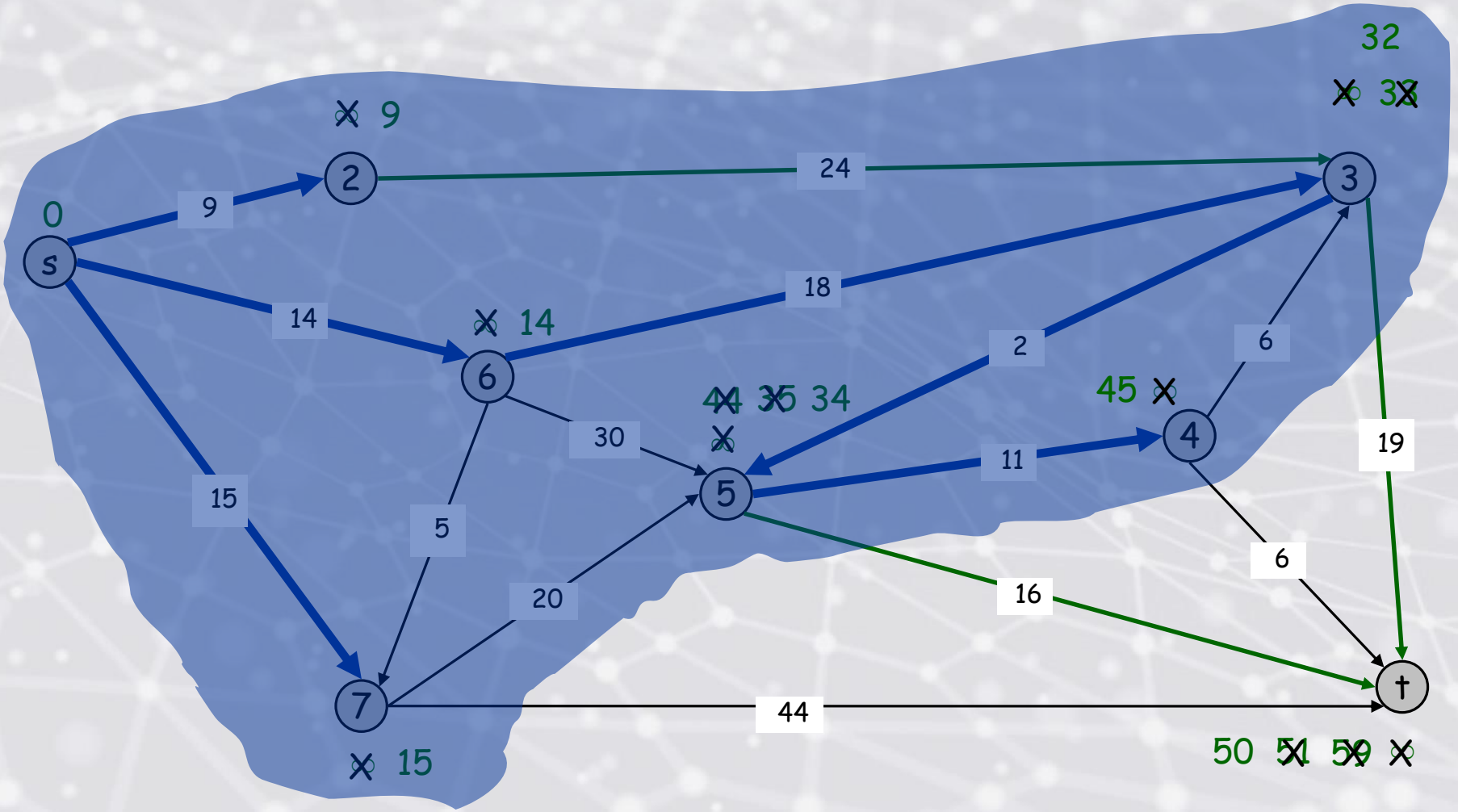
Aside: Dijkstra's Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



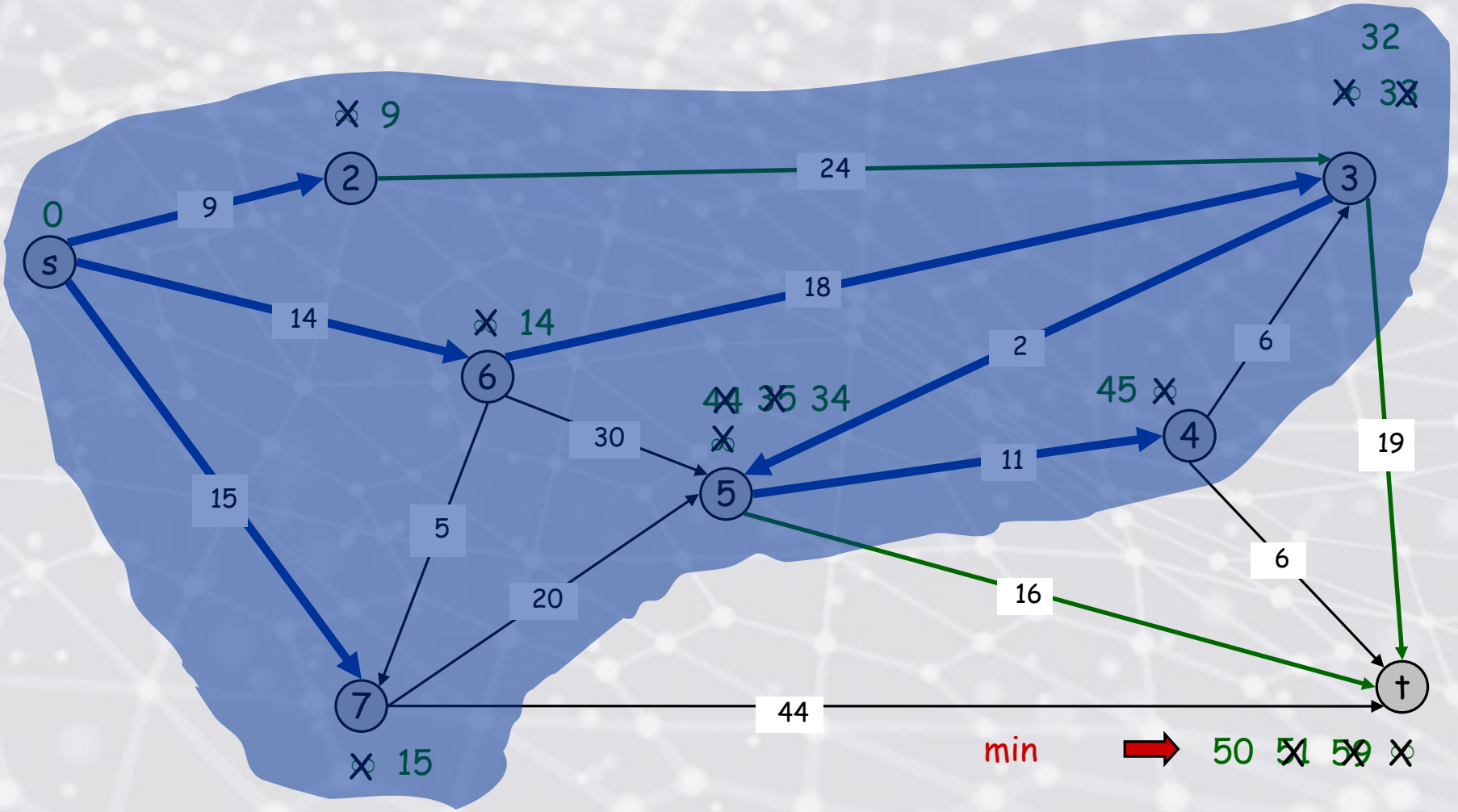
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



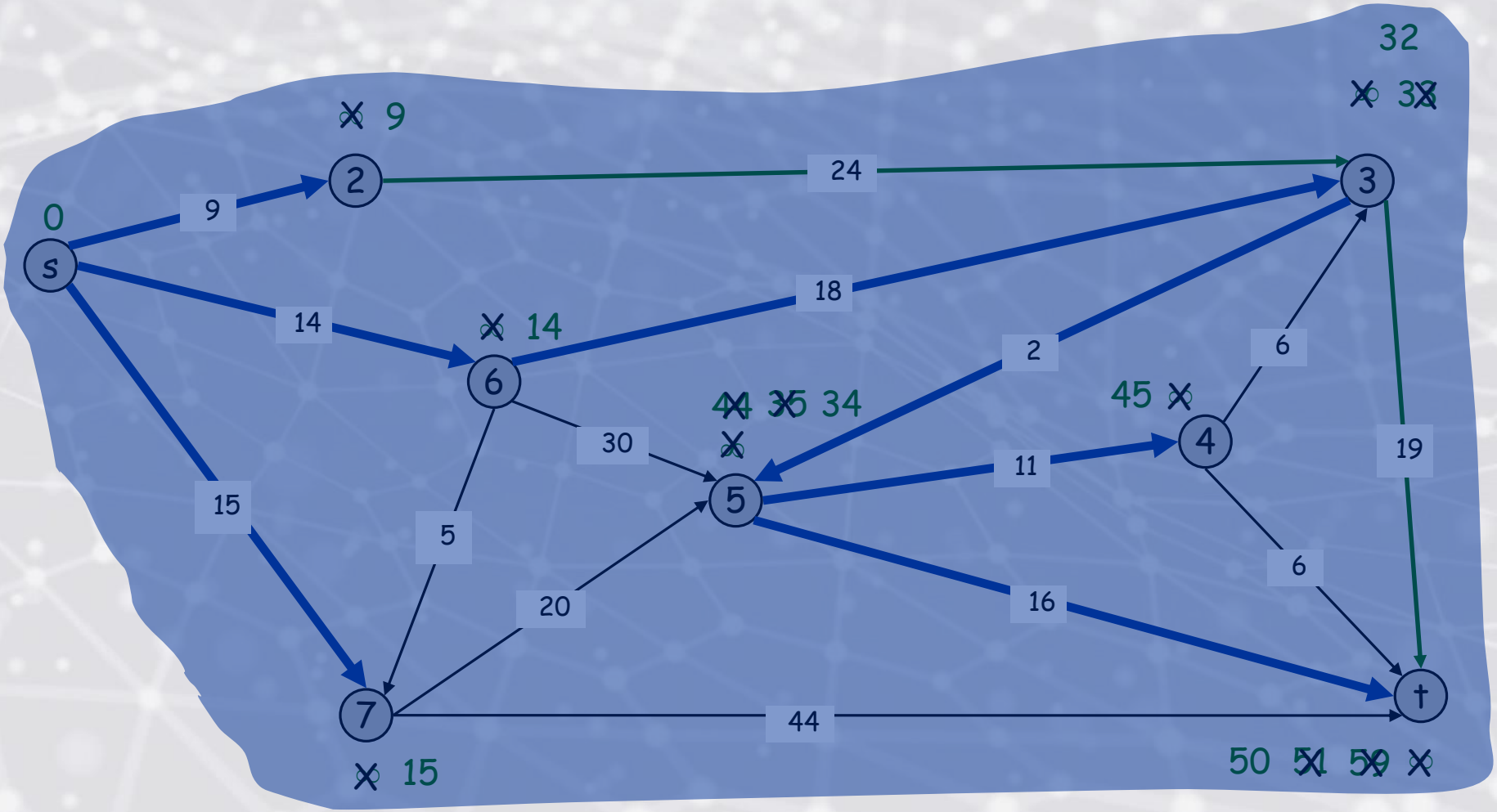
Aside: Dijkstra's Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



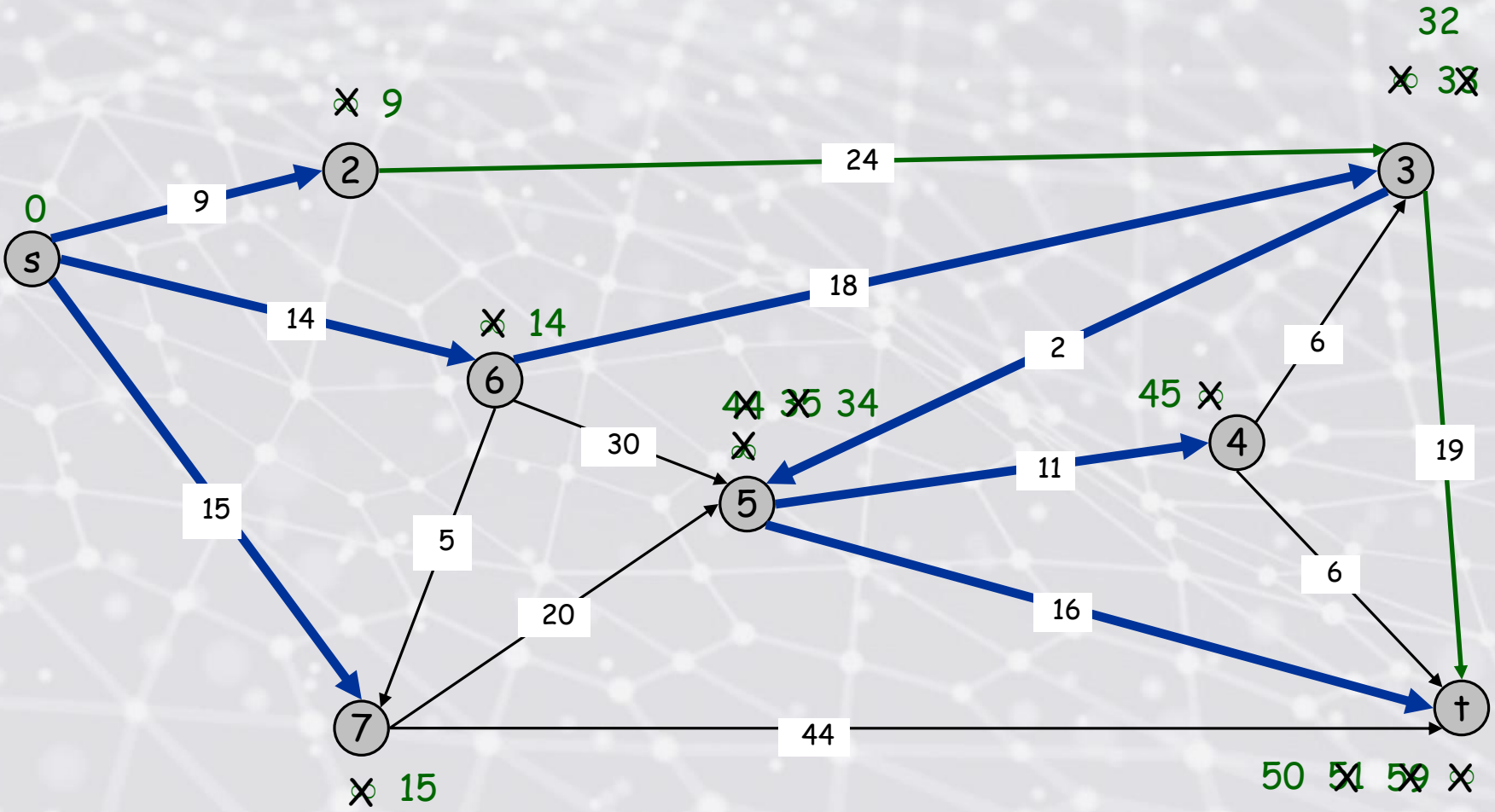
Aside: Dijkstra's Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Aside: Dijkstra's Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Aside: Dijkstra's Algorithm

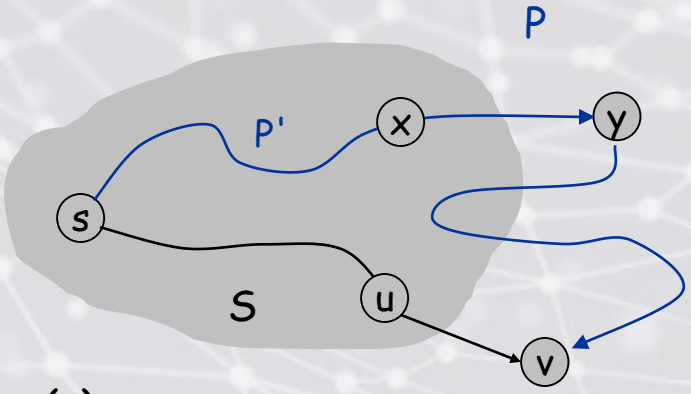
For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- (Claim) The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)$$

- Nonnegative weights (triangle inequality)
- inductive hypothesis
- defn of $\pi(y)$
- Dijkstra chose v instead of y

Aside: Dijkstra's Algorithm

For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

PQ Operation	Dijkstra	Array	Binary heap
Insert	n	n	log n
ExtractMin	n	n	log n
ChangeKey	m	1	log n
IsEmpty	n	1	1
Total		n^2	m log n

ISOMAP

(*) The Isomap algorithm is comprised of (3) general steps:

(2) In its second step, Isomap estimates the geodesic distances $d_M(i,j)$ between all pairs of points on the manifold M by computing their shortest path distances $d_G(i,j)$ in the graph G.

There are several options here: **Dijkstra's Algorithm** is perhaps the best-known (also can use *Floyd-Warshall*, for instance).

ISOMAP

(*) The Isomap algorithm is comprised of (3) general steps:

(3) The **final step** applies classical MDS to the matrix of graph distances $D_G=\{d_G(i,j)\}$, constructing an embedding of the data in a d-dimensional Euclidean space Y that best preserves the manifold's estimates of the intrinsic geometry of the data.

ISOMAP

(*) The Isomap algorithm is comprised of (3) general steps:

(3) The **final step** applies classical MDS to the matrix of graph distances $D_G=\{d_G(i,j)\}$, constructing an embedding of the data in a d-dimensional Euclidean space Y that best preserves the manifold's estimates of the intrinsic geometry of the data.

The coordinate vectors $\mathbf{y}_i$ for points in Y are chosen to minimize the cost function:

$$E = \left\| \tau(D_G) - \tau(D_Y) \right\|_2$$

• Where D_Y denotes the matrix of Euclidean distances $\{d_Y(i,j)= \|\boldsymbol{y}_i - \boldsymbol{y}_j\| \}$ and $\|\cdot\|_2$ in the equation above is the L2 matrix norm. The τ operator converts distances to inner products which uniquely characterizes the geometry of the data in a form that supports efficient optimization.

The global minimum of the equation above is achieved by setting the coordinates $\mathbf{y}_i$ to the top *d* eigenvectors of the matrix $\tau(D_G)$.

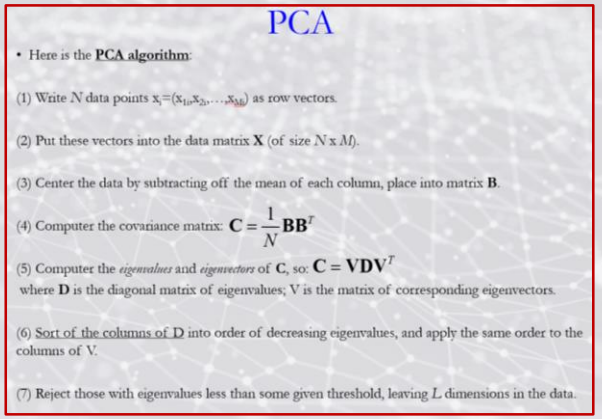
ISOMAP

(*) The Isomap algorithm is comprised of (3) general steps:

(3) The final step applies classical MDS to the matrix of graph distances $D_G=\{d_G(i,j)\}$, constructing an embedding of the data in a d-dimensional Euclidean space Y that best preserves the manifold's estimates intrinsic geometry.

(*) This third step in the Isomap algorithm might seem somewhat complicated – however, it is really equivalent to performing PCA on the (centered) *square* of a distance matrix D (for a data set).

• In other words, you take the distance matrix D, square it, and center. Then compute the eigendecomposition for the top eigenvalues and project the data into this smaller dimensional space (just as with PCA).



ISOMAP

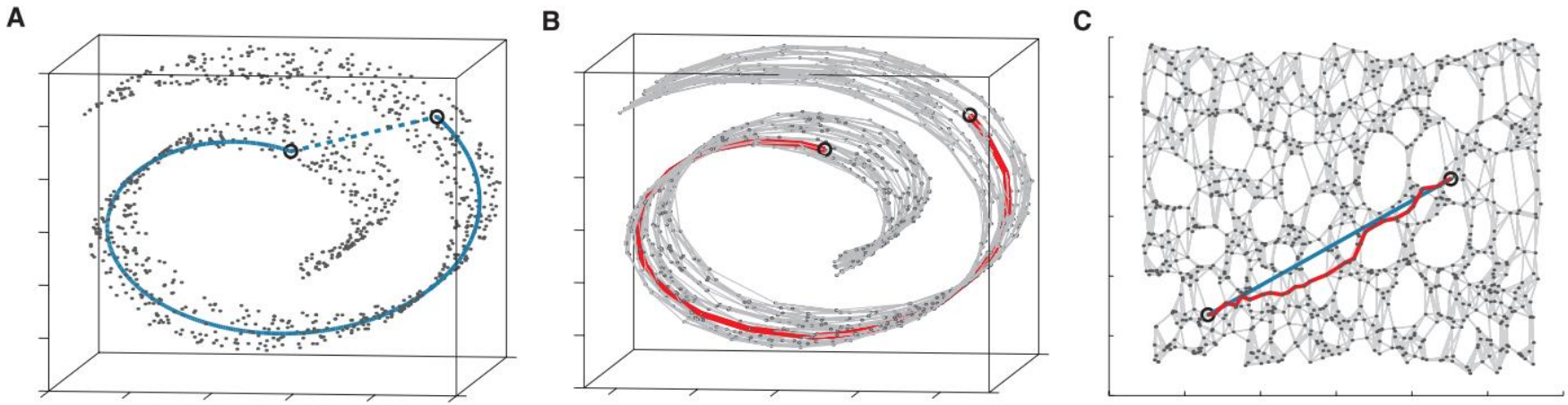


Fig. 3. The "Swiss roll" data set, illustrating how Isomap exploits geodesic paths for nonlinear dimensionality reduction. (**A**) For two arbitrary points (circled) on a nonlinear manifold, their Euclidean distance in the high-dimensional input space (length of dashed line) may not accurately reflect their intrinsic similarity, as measured by geodesic distance along the low-dimensional manifold (length of solid curve). (**B**) The neighborhood graph G constructed in step one of Isomap (with $K = 7$ and $N =$ 1000 data points) allows an approximation (red segments) to the true geodesic path to be computed efficiently in step two, as the shortest path in G. (**C**) The two-dimensional embedding recovered by Isomap in step three, which best preserves the shortest path distances in the neighborhood graph (overlaid). Straight lines in the embedding (blue) now represent simpler and cleaner approximations to the true geodesic paths than do the corresponding graph paths (red).

Potential Issues for Isomap: The connectivity of each data point in the neighborhood graph is defined as its nearest k Euclidean neighbors in the high-dimensional space. This step is vulnerable to "short-circuit errors" if k is too large with respect to the manifold structure or if noise in the data moves the points slightly off the manifold. Even a single short-circuit error can alter many entries in the geodesic distance matrix, which in turn can lead to a drastically different (and incorrect) low-dimensional embedding. Conversely, if k is too small, the neighborhood graph may become too sparse to approximate geodesic paths accurately. But improvements have been made to this algorithm to make it work better for sparse and noisy data sets.

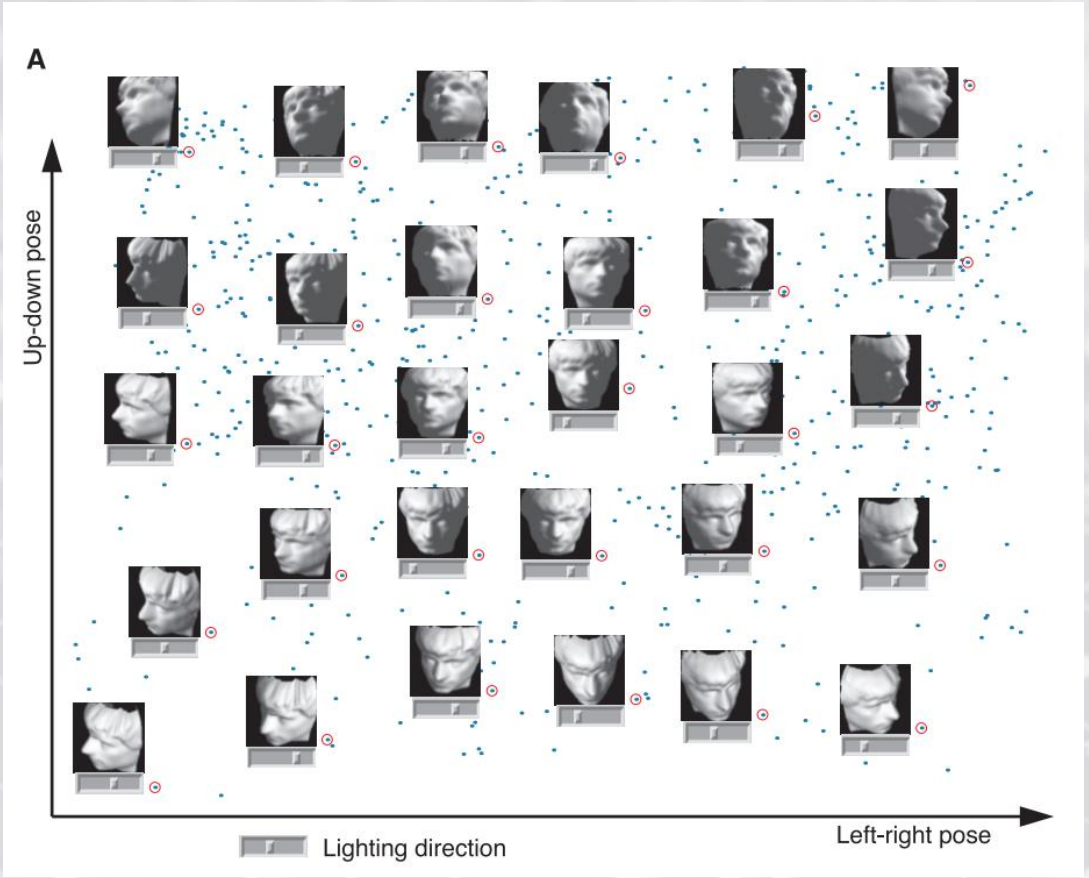
ISOMAP

Table 1. The Isomap algorithm takes as input the distances $d_X(i,j)$ between all pairs i,j from N data points in the high-dimensional input space X, measured either in the standard Euclidean metric (as in Fig. 1A) or in some domain-specific metric (as in Fig. 1B). The algorithm outputs coordinate vectors $\mathbf{y}_i$ in a d-dimensional Euclidean space Y that (according to Eq. 1) best represent the intrinsic geometry of the data. The only free parameter (ϵ or K) appears in Step 1.

Step		
1	Construct neighborhood graph	Define the graph G over all data points by connecting points i and j if [as measured by $d_X(i,j)$] they are closer than ϵ (ϵ-Isomap), or if i is one of the K nearest neighbors of j (K-Isomap). Set edge lengths equal to $d_X(i,j)$.
2	Compute shortest paths	Initialize $d_G(i,j) = d_X(i,j)$ if i,j are linked by an edge; $d_G(i,j) = \infty$ otherwise. Then for each value of $k = 1, 2, \ldots, N$ in turn, replace all entries $d_G(i,j)$ by $\min\{d_G(i,j), d_G(i,k) + d_G(k,j)\}$. The matrix of final values $D_G = \{d_G(i,j)\}$ will contain the shortest path distances between all pairs of points in G (*16*, *19*).
3	Construct d-dimensional embedding	Let λ_p be the p-th eigenvalue (in decreasing order) of the matrix $\tau(D_G)$ (*17*), and v_p^i be the i-th component of the p-th eigenvector. Then set the p-th component of the d-dimensional coordinate vector $\mathbf{y}_i$ equal to $\sqrt{\lambda_p} v_p^i$.

ISOMAP

Fig. 1. (**A**) A canonical dimensionality reduction problem from visual perception. The input consists of a sequence of 4096-dimensional vectors, representing the brightness values of 64 pixel by 64 pixel images of a face rendered with different poses and lighting directions. Applied to $N = 698$ raw images, Isomap ($K = 6$) learns a three-dimensional embedding of the data's intrinsic geometric structure. A two-dimensional projection is shown, with a sample of the original input images (red circles) superimposed on all the data points (blue) and horizontal sliders (under the images) representing the third dimension. Each coordinate axis of the embedding correlates highly with one degree of freedom underlying the original data: left-right pose (x axis, $R = 0.99$), up-down pose (y axis, $R = 0.90$), and lighting direction (slider position, $R = 0.92$). The input-space distances $d_X(i,j)$ given to Isomap were Euclidean distances between the 4096-dimensional image vectors. (**B**)



ISOMAP

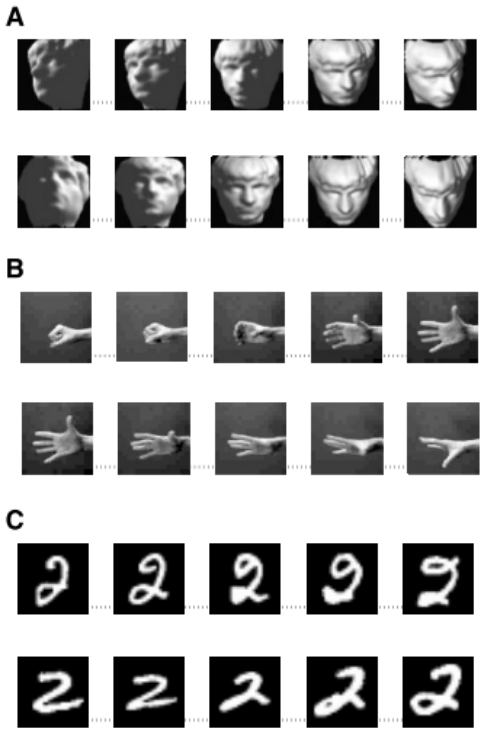
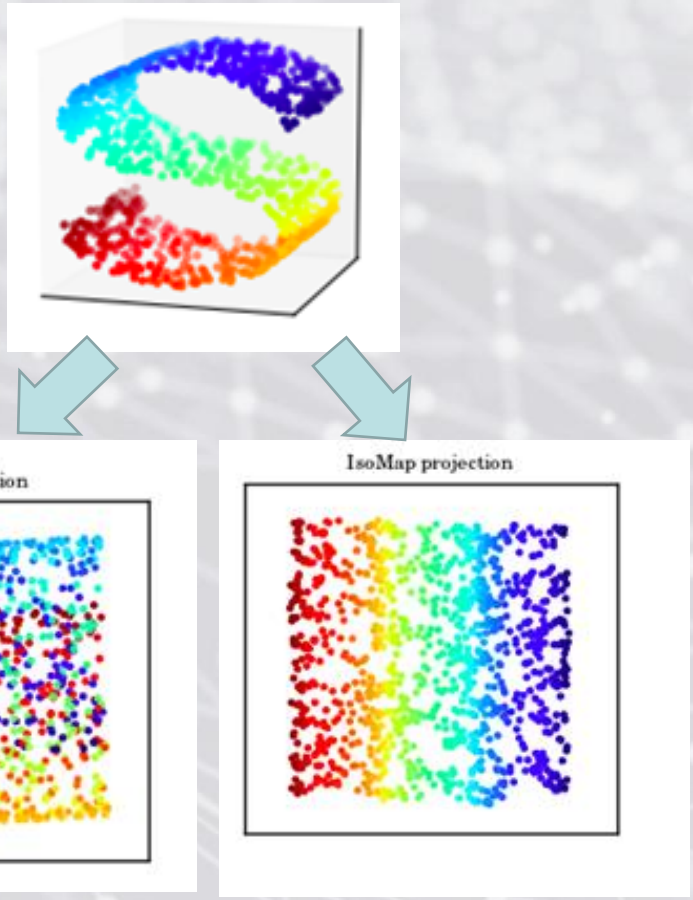


Fig. 4. Interpolations along straight lines in the Isomap coordinate space (analogous to the blue line in Fig. 3C) implement perceptually natural but highly nonlinear "morphs" of the corresponding high-dimensional observations (*43*) by transforming them approximately along geodesic paths (analogous to the solid curve in Fig. 3A). (**A**) Interpolations in a three-dimensional embedding of face images (Fig. 1A). (**B**) Interpolations in a four-dimensional embedding of hand images (*20*) appear as natural hand movements when viewed in quick succession, even though no such motions occurred in the observed data. (**C**) Interpolations in a six-dimensional embedding of handwritten "2"s (Fig. 1B) preserve continuity not only in the visual features of loop and arch articulation, but also in the implied pen trajectories, which are the true degrees of freedom underlying those appearances.



Python implementation:
http://scikit-learn.org/stable/modules/manifold.html

Fin