CSE 5526: Introduction to Neural Networks

Second Half Review

Main topics covered in first half of class

- McCulloch-Pitts neurons
  - Designing networks by hand
  - Training using the perceptron algorithm
- Linear regression
  - Closed-form solution
  - Training using gradient descent
- Multi-layer perceptrons
  - Backpropagation training algorithm
  - Generalization, over-fitting, under-fitting, learning curves
- Radial basis function networks

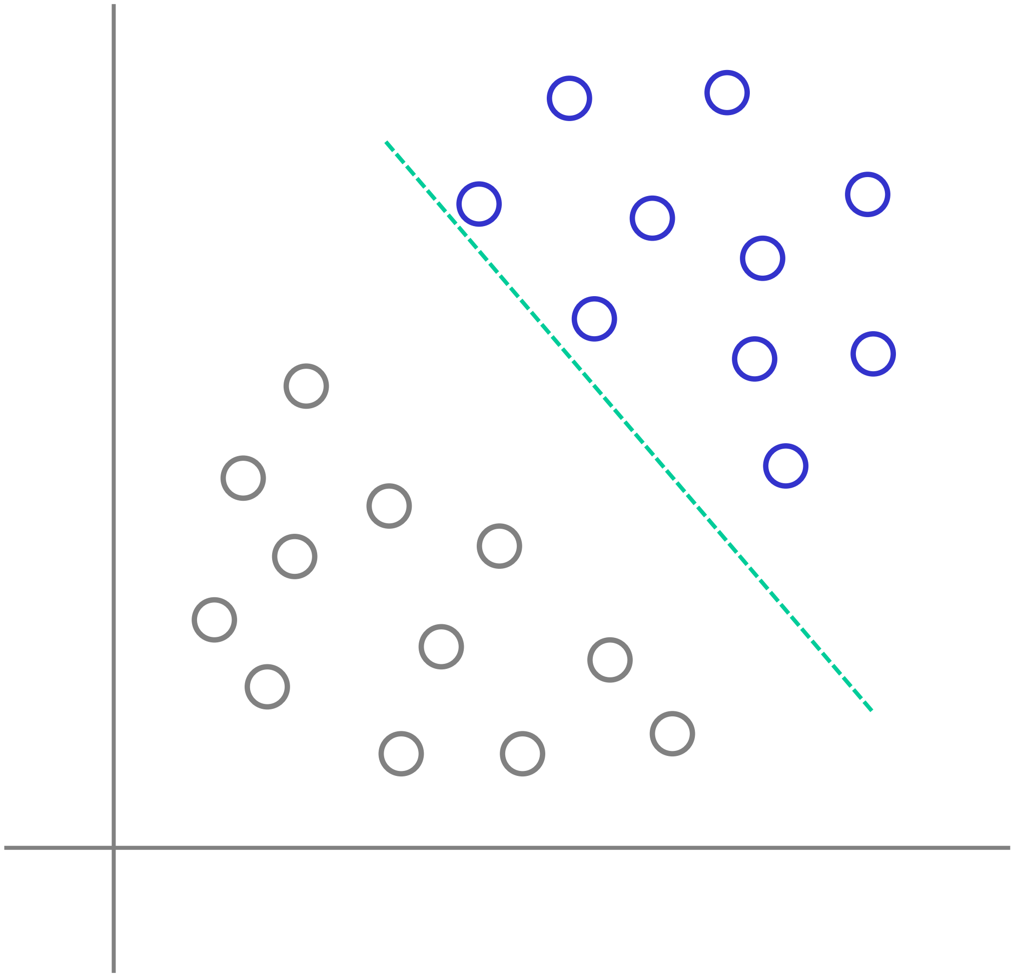
Main topics covered in second half of class

- Support vector machines
  - Lagrange multipliers
  - Maximum margin formulation, primal and dual
  - Kernels
  - Training SVMs on non-separable data (slack variables)
- Unsupervised learning
  - Self-organizing maps
  - Hopfield networks
  - (Restricted) Boltzmann machines
  - Deep belief networks and deep neural networks

Support Vector Machines (SVM)

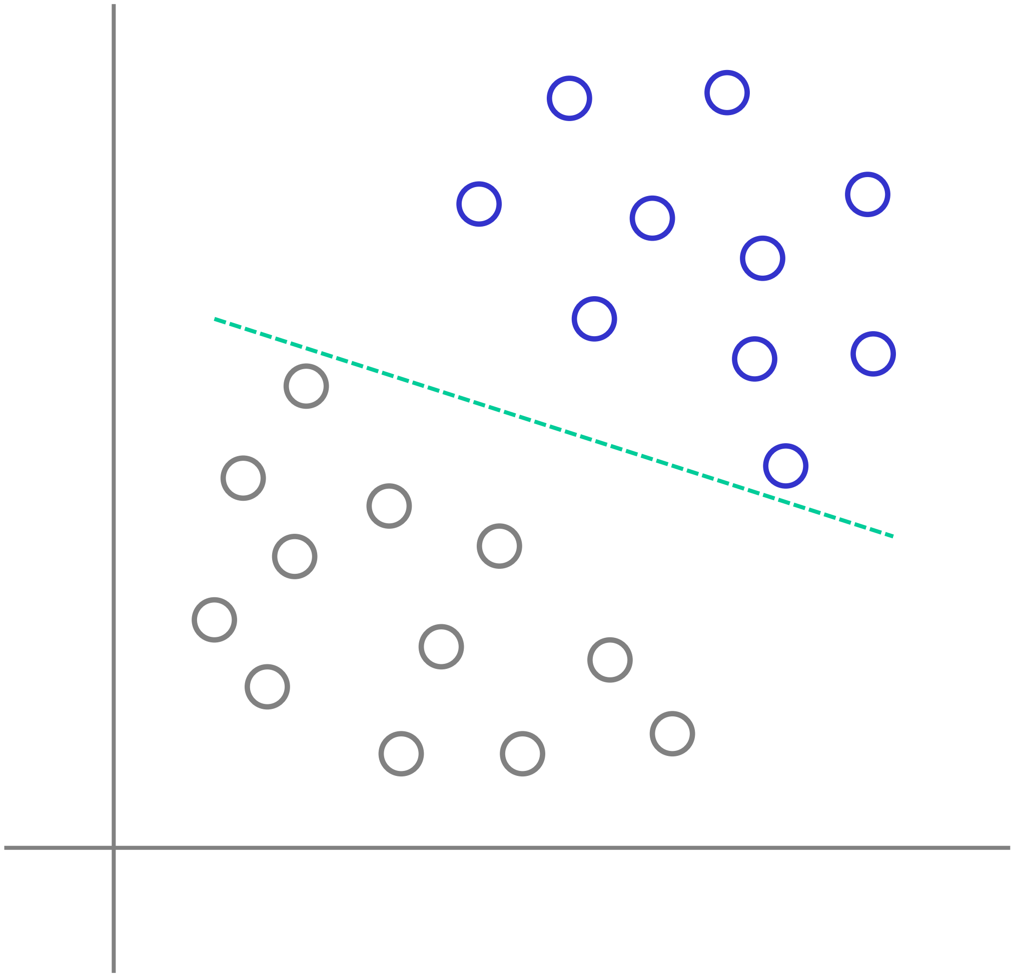
Perceptrons find any separating hyperplane

Depends on initialization and ordering of training points



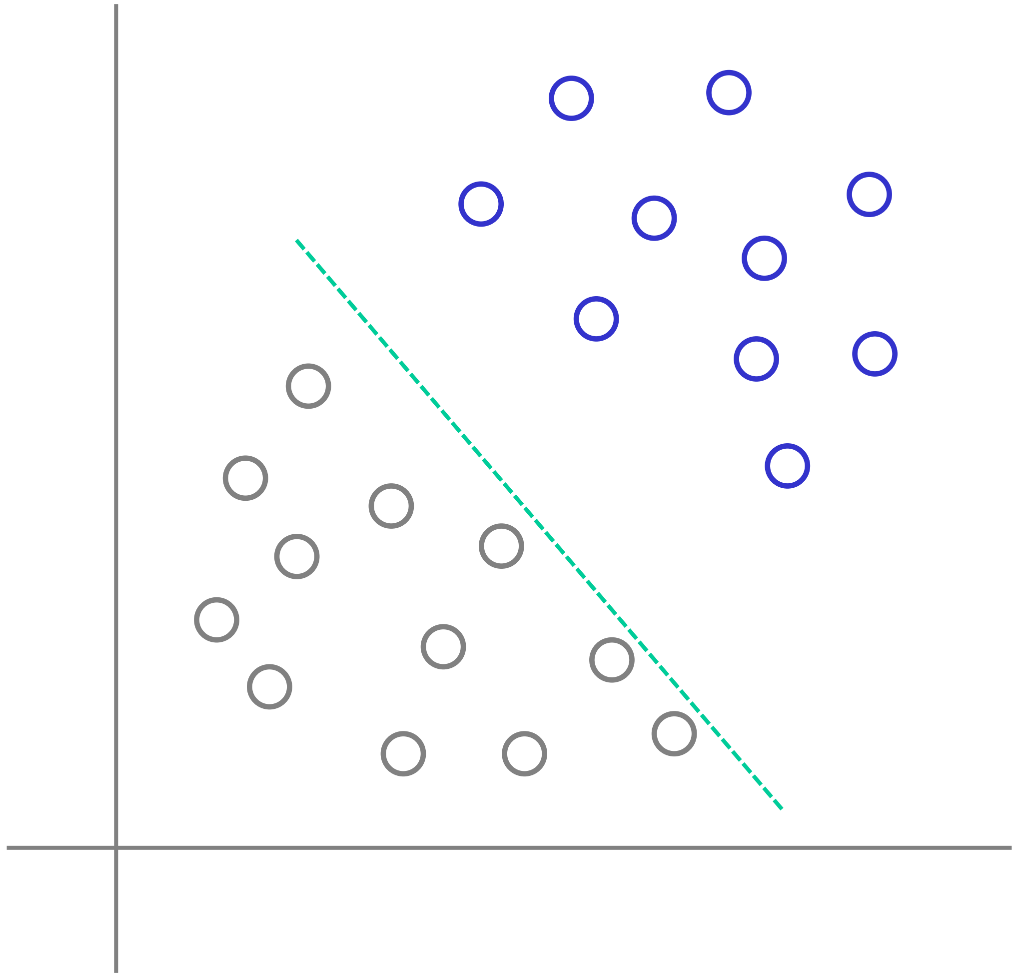
Perceptrons find any separating hyperplane

Depends on initialization and ordering of training points



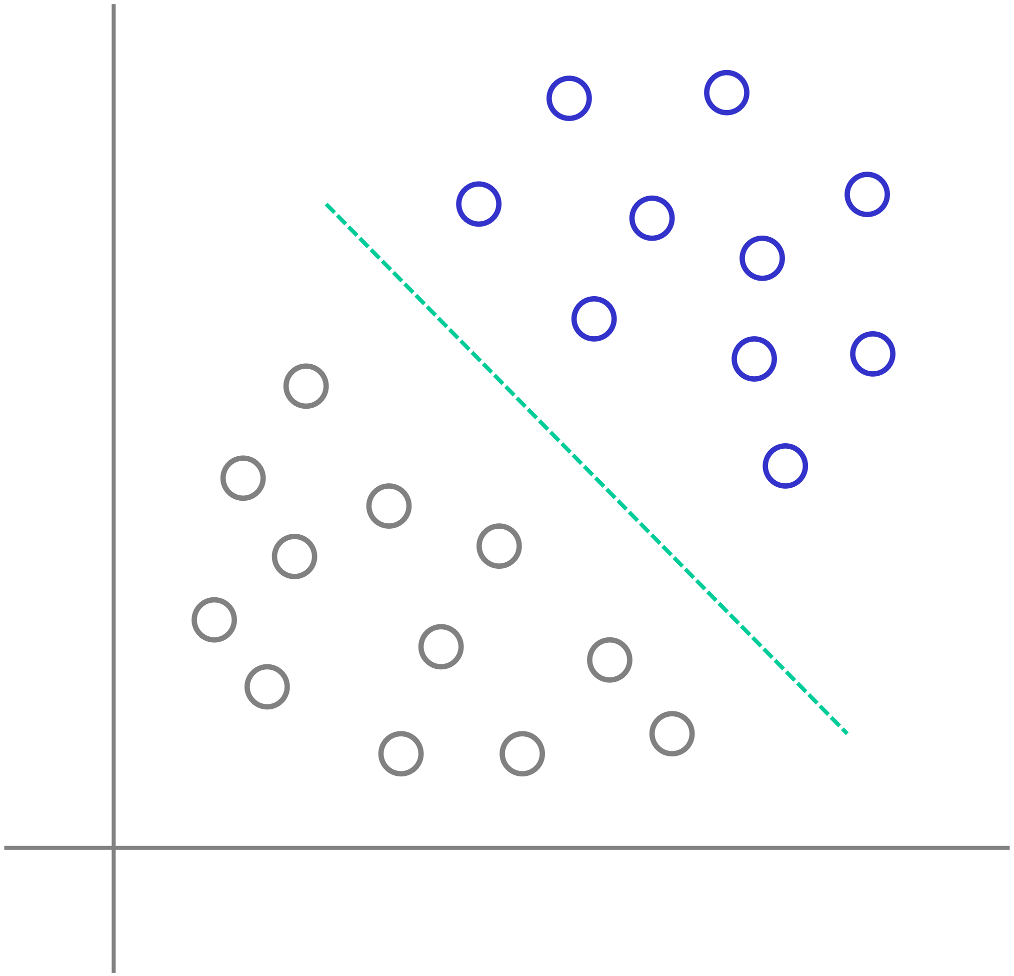
Perceptrons find any separating hyperplane

Depends on initialization and ordering of training points



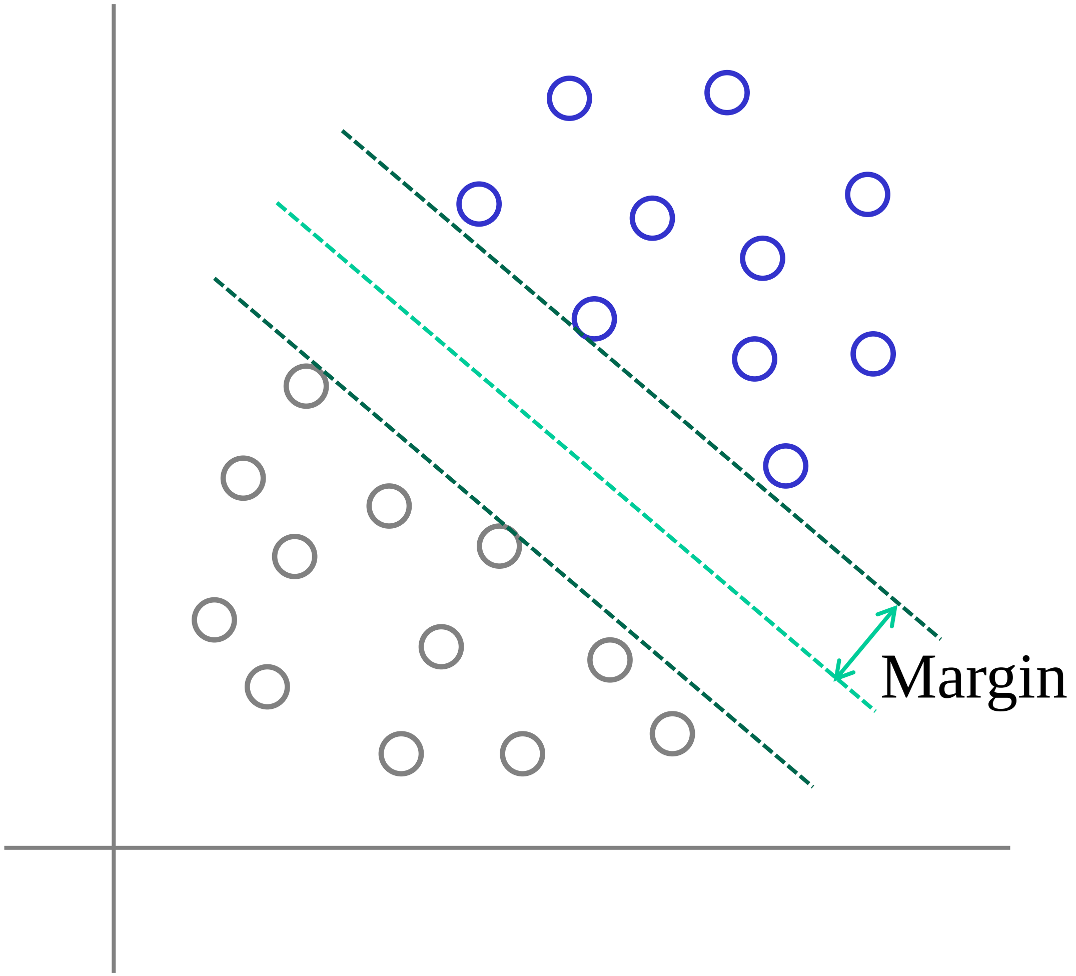
Perceptrons find any separating hyperplane

Depends on initialization and ordering of training points

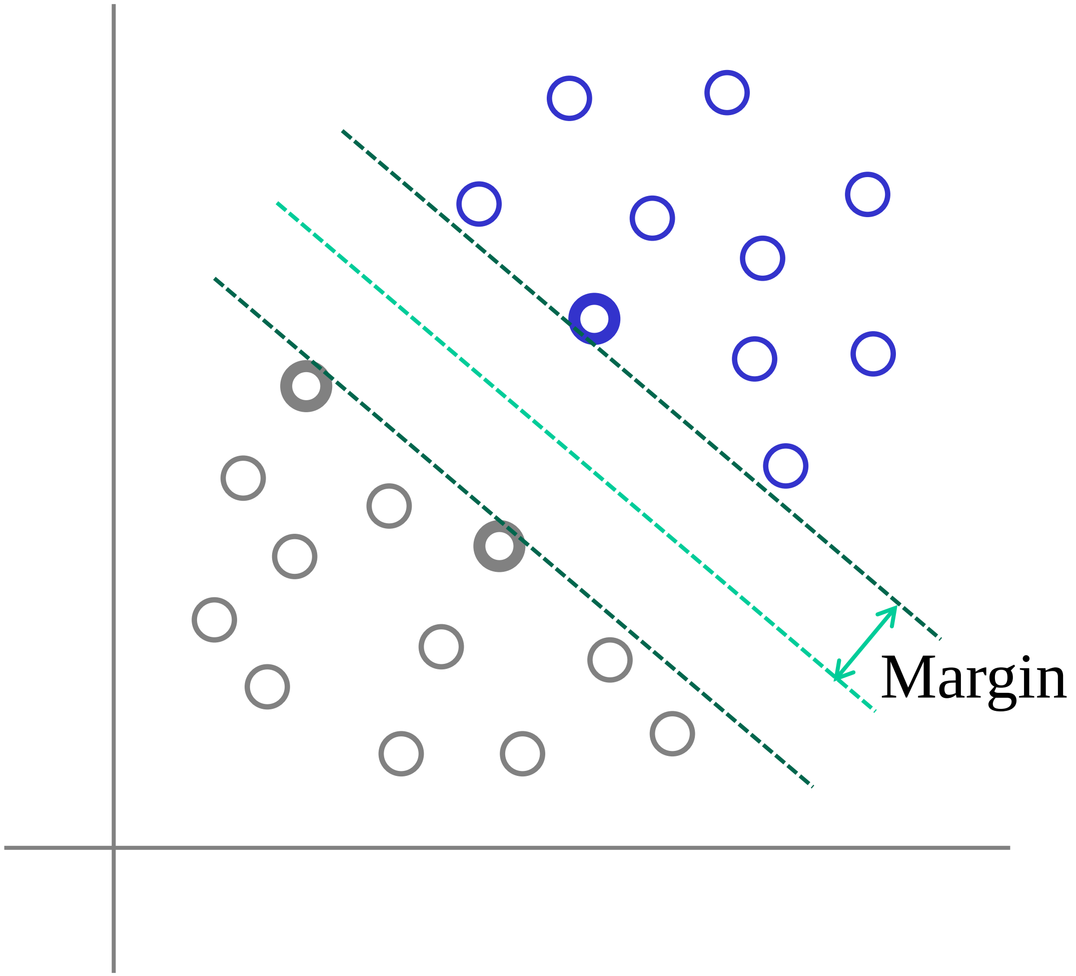


But the maximum margin hyperplane generalizes the best to new data

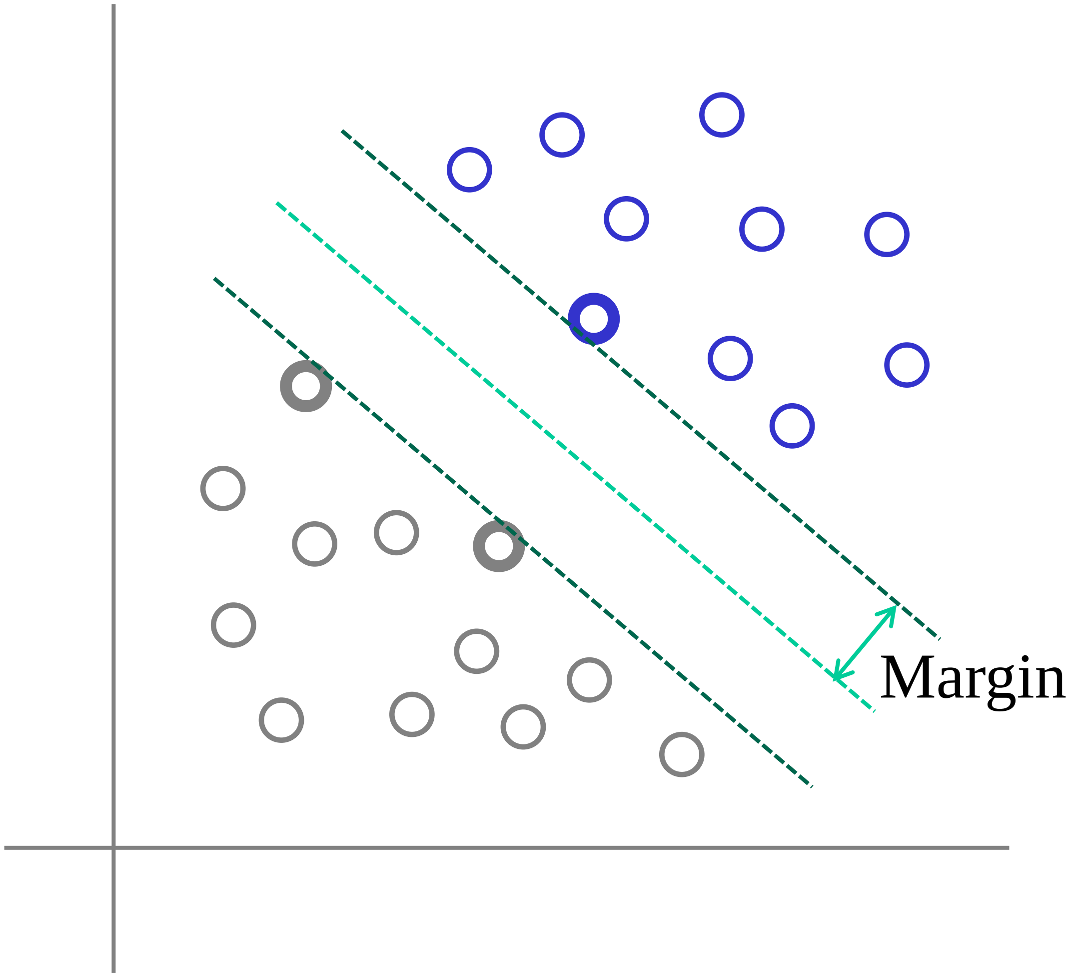
According to computational/statistical learning theory



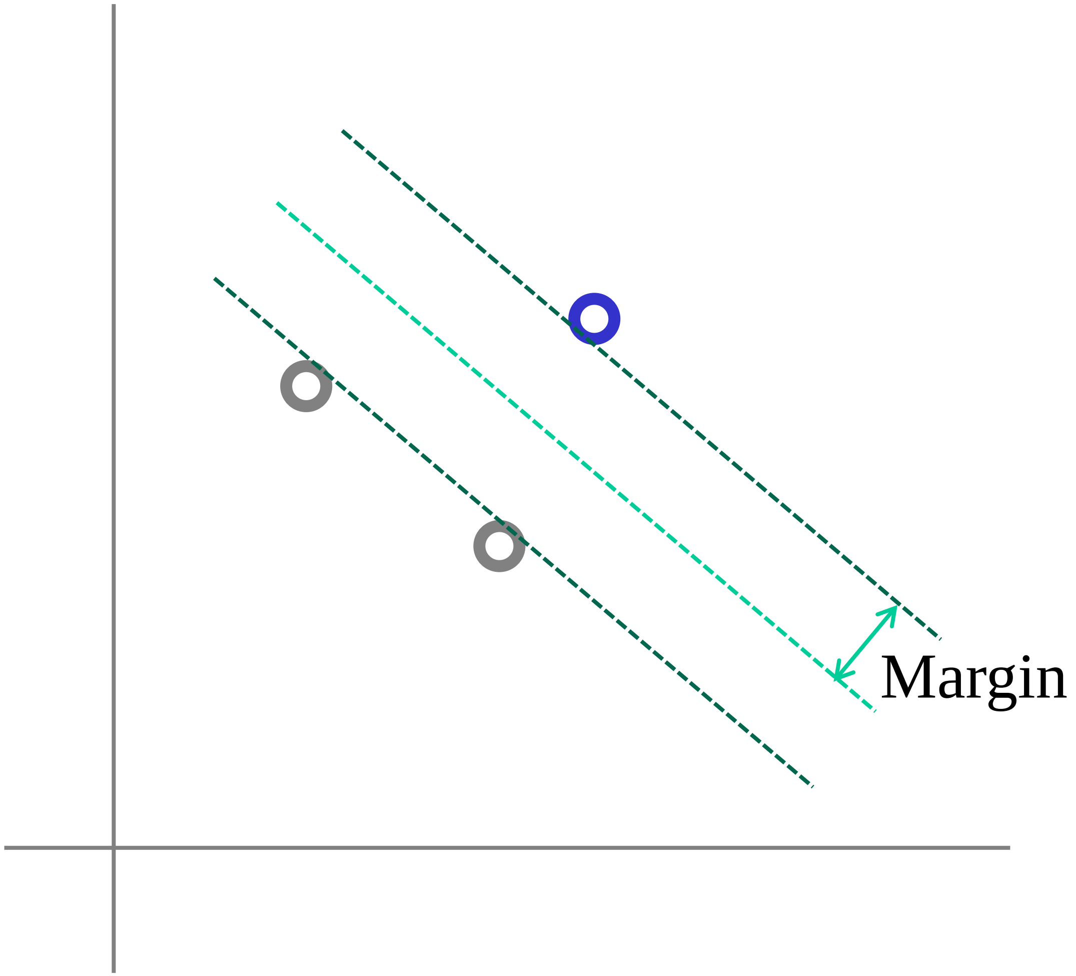
The maximum margin only depends on certain points, the support vectors



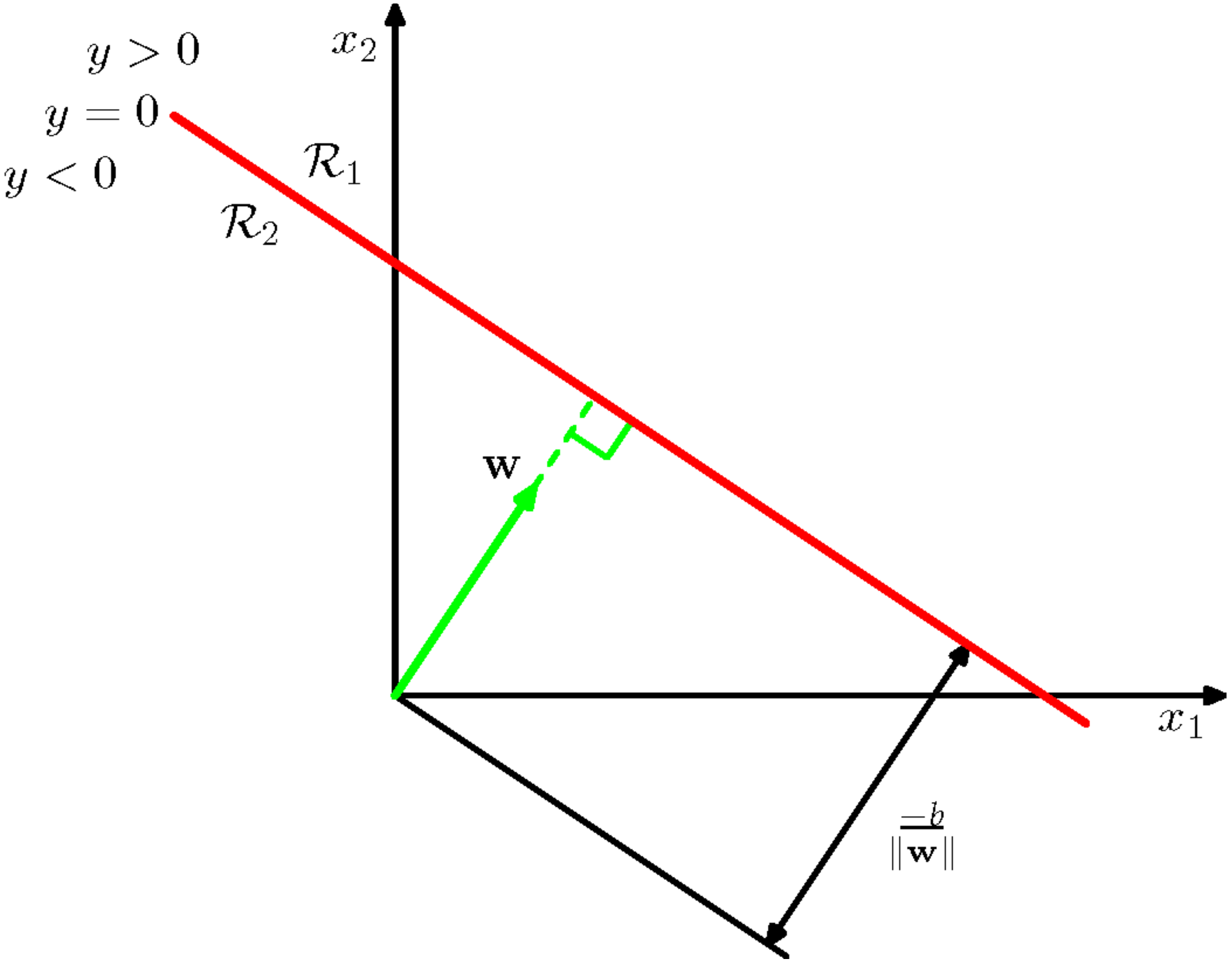
The maximum margin only depends on certain points, the support vectors



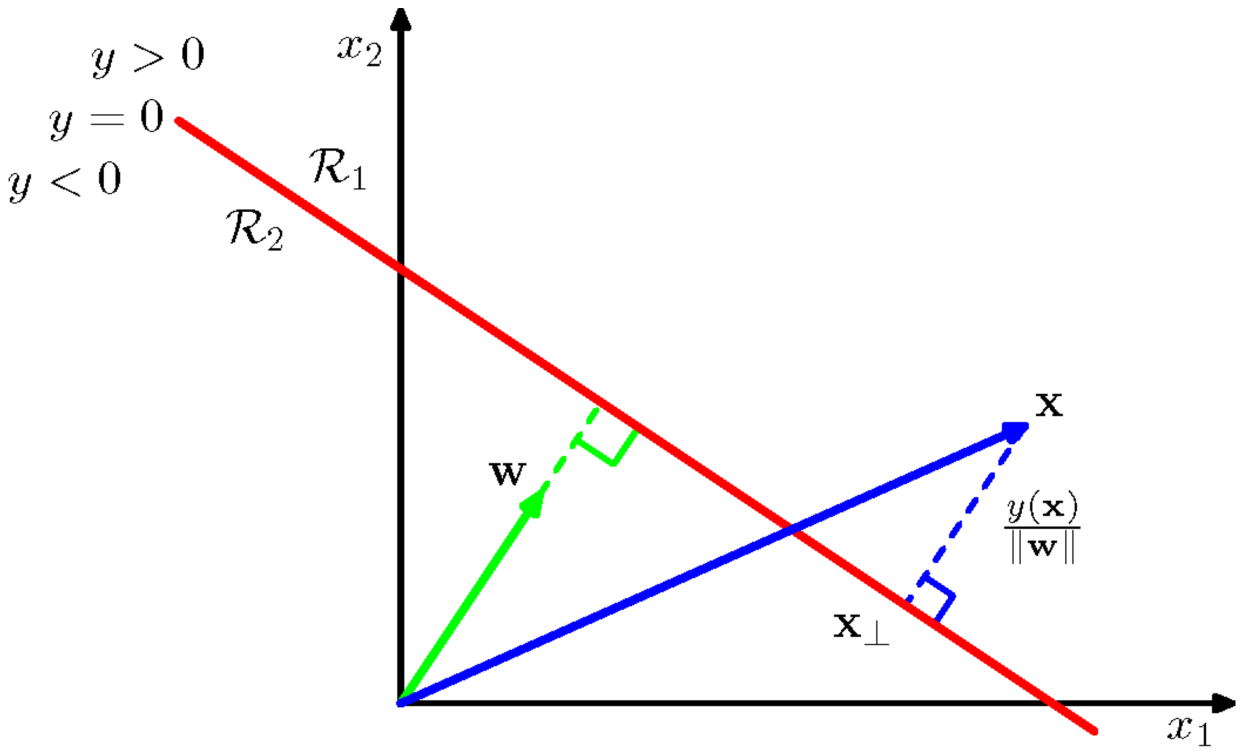
The maximum margin only depends on certain points, the support vectors



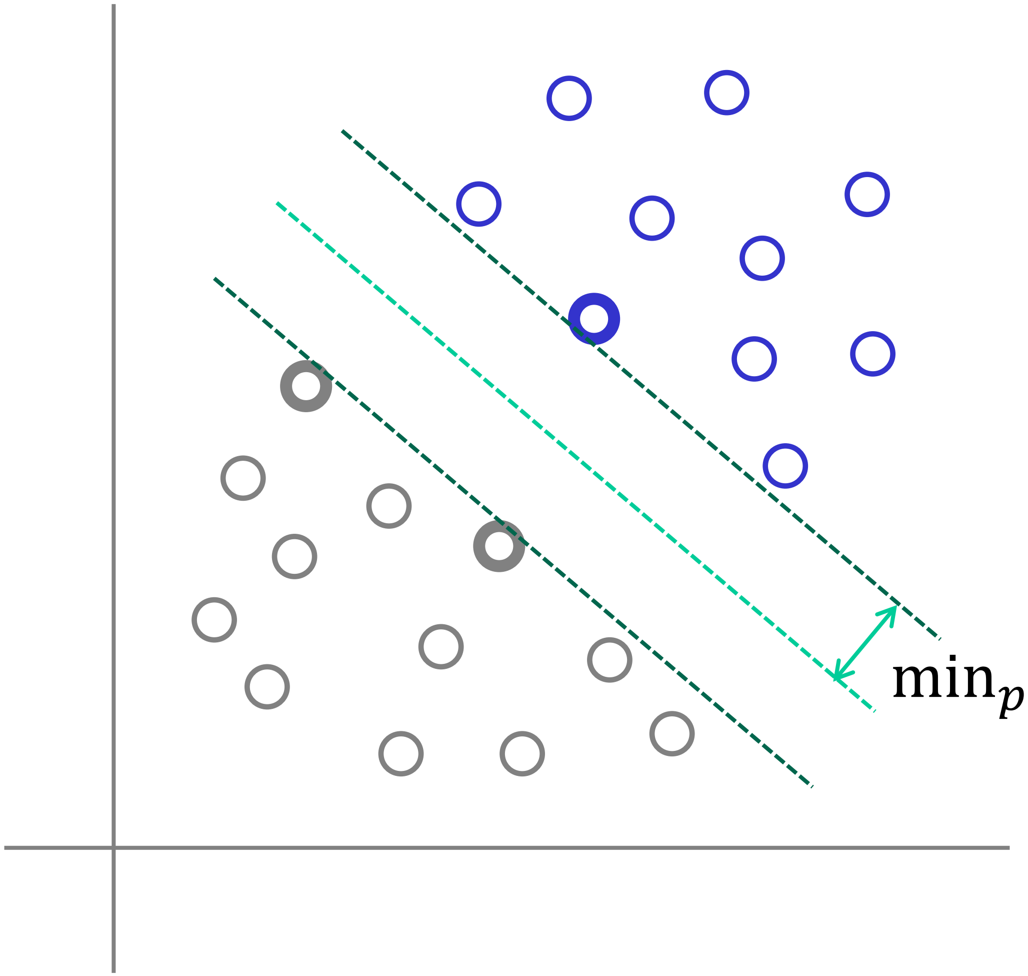
w is perpendicular to the hyperplane, *b* defines its distance from the origin



The distance from point $\boldsymbol{x}$ to the hyperplane is $y(\boldsymbol{x})/\|\boldsymbol{w}\|$



The maximum margin hyperplane is farthest from all of the data points



$\min_p d_p y(\boldsymbol{x}_p)/\|\boldsymbol{w}\|$

Maximum margin constrained optimization problem

- Which is equivalent to

$$\text{argmin}_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 \text{ subject to } d_p\left(\boldsymbol{w}^T\boldsymbol{x}_p + b\right) \geq 1$$

- This is a well studied type of problem
  - A quadratic program with linear inequality constraints

Detour: Lagrange multipliers solve constrained optimization problems

- Want to maximize a function $f(x_1, x_2)$
- Subject to the equality constraint $g(x_1, x_2) = 0$
- Could solve $g(x_1, x_2) = 0$ for x_1 in terms of x_2
  - But that is hard to do in general (i.e., on computers)
- Or could use Lagrange multipliers
  - Which are easier to use in general (i.e., on computers)

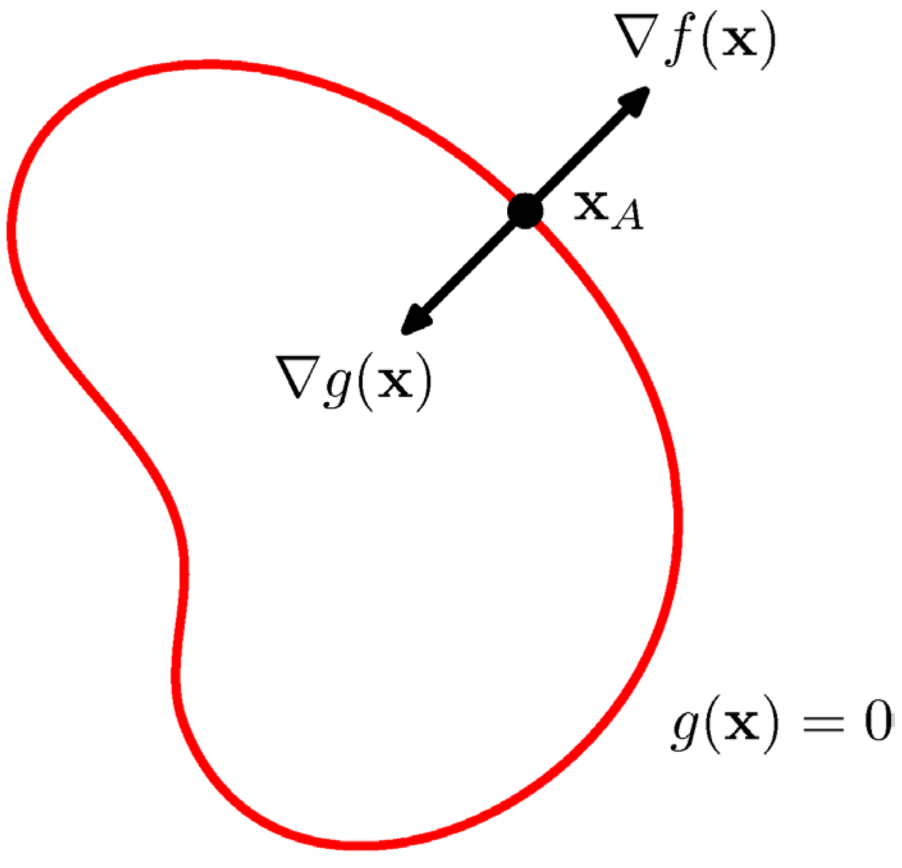
Lagrange multipliers with general $\boldsymbol{x}$

- In general, we can write

$$\max_{\boldsymbol{x}} f(\boldsymbol{x}) \text{ subject to } g(\boldsymbol{x}) = 0$$

- Constraint $g(\boldsymbol{x}) = 0$ defines a $D - 1$ dimensional surface for D dimensional $\boldsymbol{x}$

Gradients of g and f are orthogonal to surface at solution point



Gradients of g and f are orthogonal to surface at maximum of f

- For g because for all points on the surface $g(\boldsymbol{x}) = 0$
  - Meaning that the directional derivative along it is 0
  - So the gradient must be perpendicular to it
- For f because if it wasn't, you could move along the surface in the direction of the gradient to find a better maximum of f
- Thus ∇f and ∇g are (anti-)parallel
- And there must exist a scalar λ such that

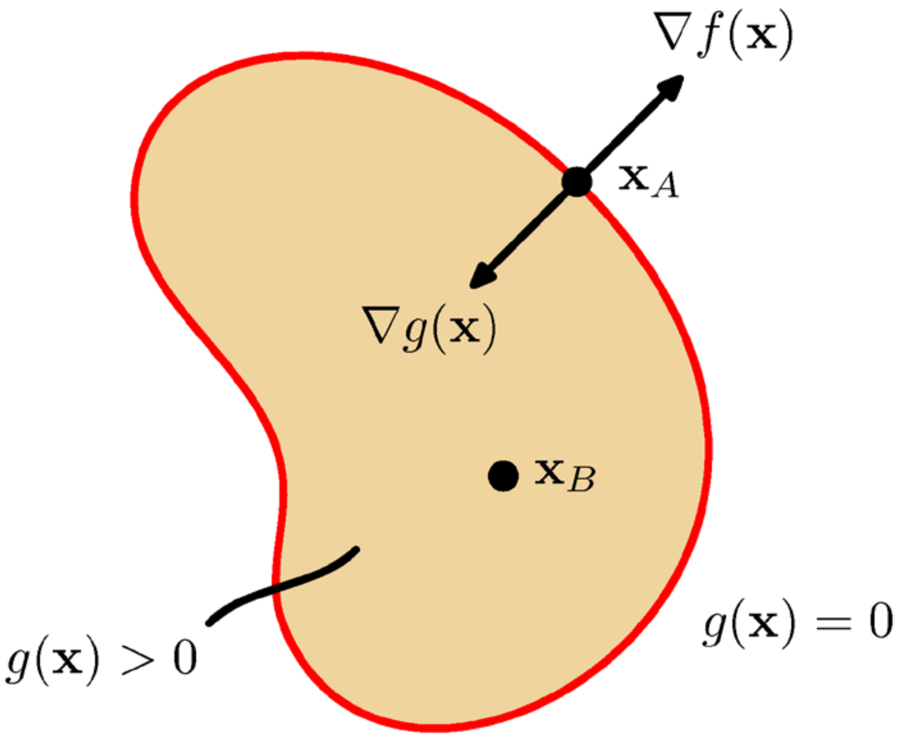
$$\nabla f + \lambda \nabla g = 0$$

The Lagrangian function captures the constraints on x and on the gradients

$$L(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda g(\boldsymbol{x})$$

- Setting gradient of L with respect to $\boldsymbol{x}$ to 0 gives
$$\nabla f + \lambda \nabla g = 0$$
- Setting partial of L with respect to λ to 0 gives
$$g(\boldsymbol{x}) = 0$$
- Thus stationary points of L solve the constrained optimization problem

Lagrange multipliers can also be used with inequality constraints $g(\boldsymbol{x}) \geq 0$



Back to SVMs: Maximum margin solution is a fixed point of the Lagrangian function

- Recall, the maximum margin hyperplane is

$$\text{argmin}_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 \quad \text{subject to } d_p\left(\boldsymbol{w}^T\boldsymbol{x}_p + b\right) \geq 1$$

  - Minimization of a quadratic function subject to multiple linear inequality constraints
- Will use Lagrange multipliers, a_p, to write Lagrangian function

$$L(\boldsymbol{w}, b, \boldsymbol{a}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_p a_p(d_p\left(\boldsymbol{w}^T\boldsymbol{x}_p + b\right) - 1)$$

  - Note that $\boldsymbol{x}_p$ and d_p are fixed for the optimization

Dual form of Lagrangian eliminates $\boldsymbol{w}$ and b

- Dual form of Lagrangian, maximize:

$$\tilde{L}(\boldsymbol{a}) = -\frac{1}{2}\sum_p \sum_q a_p a_q \, d_p d_q \boldsymbol{x}_p^T \boldsymbol{x}_q + \sum_p a_p$$

- Subject to the constraints

$$a_p \geq 0 \;\; \forall p \qquad\qquad \sum_p a_p d_p = 0$$

- Another quadratic programming problem subject to linear inequality and equality constraints

Classify new points using $y(\boldsymbol{x})$

- Actual prediction function is still

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + b$$

- Get $\boldsymbol{w}$ from primal Lagrangian

$$\boldsymbol{w} = \sum_p a_p d_p \boldsymbol{x}_p$$

- Will discuss b shortly, so

$$y(\boldsymbol{x}) = \sum_p a_p d_p \boldsymbol{x}_p^T \boldsymbol{x} + b$$

Classify new points using $y(\boldsymbol{x})$, with kernel

- With a kernel, $\boldsymbol{w}^T = \sum_p a_p d_p \phi(\boldsymbol{x}_p)$
- Actual prediction function is

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + b$$

$$= \sum_p a_p d_p \phi^T(\boldsymbol{x}_p) \phi(\boldsymbol{x}) + b$$

$$= \sum_p a_p d_p k(\boldsymbol{x}_p, \boldsymbol{x}) + b$$

- In practice, save all $\boldsymbol{x}_p$ with $a_p > 0$
  - And compute $k(\boldsymbol{x}_p, \boldsymbol{x})$ at test time

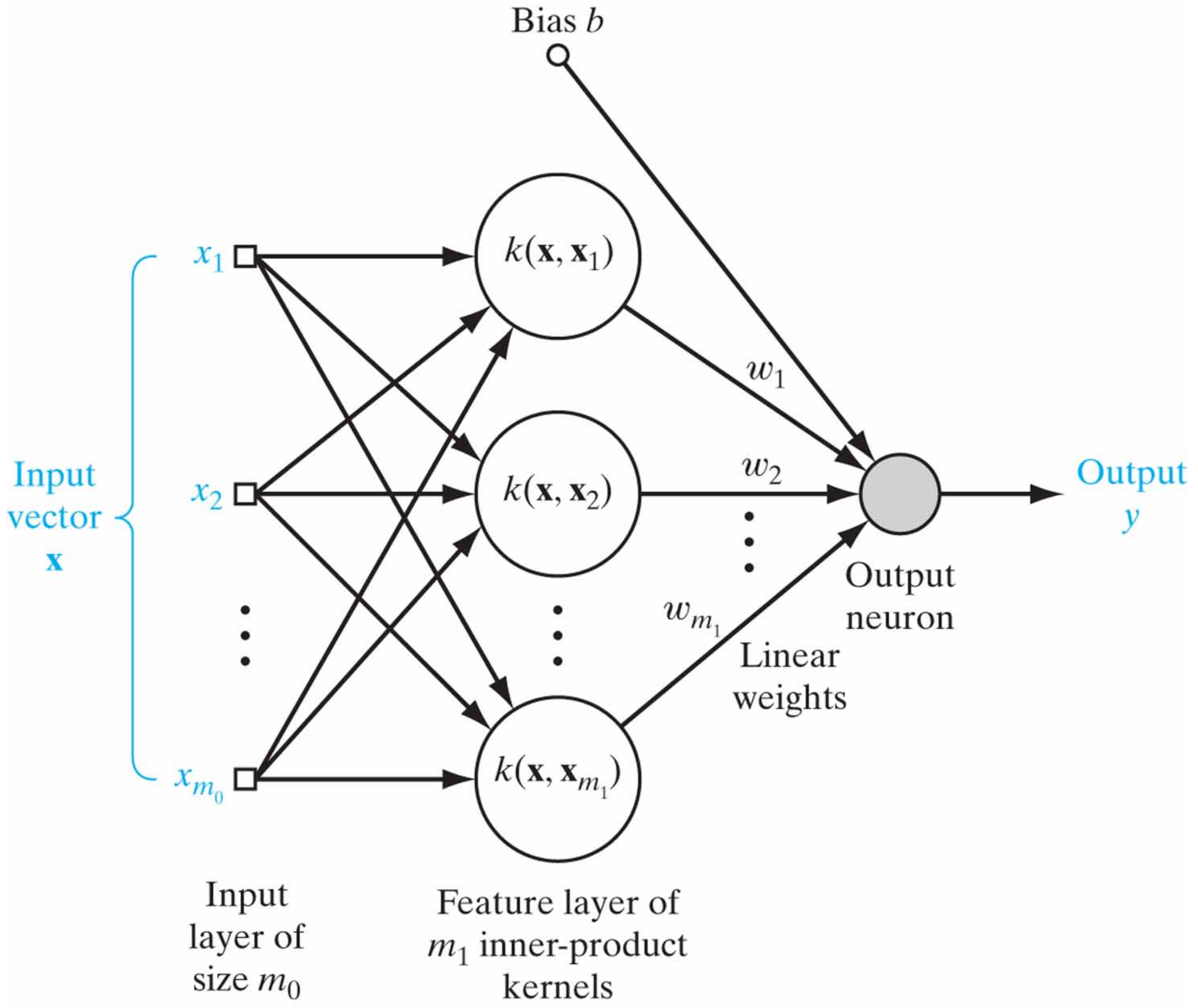
Summary so far

- Finding the maximum margin hyperplane has been formulated as a constrained quadratic program
  - Convex problem, well studied, easy conceptually to solve
- Can be solved in the primal or dual formulation
  - Dual formulation permits the use of kernel functions
- Only some data points contribute to the solution
  - The support vectors
- So far, only applies to linearly separable data

Kernels are generalizations of inner products

- A kernel is a function of two data points such that
$$k(x, x') = \phi^T(x)\phi(x')$$
For some function $\phi(x)$
- It is therefore symmetric: $k(x, x') = k(x', x)$
- Can compute $k(x, x')$ from an explicit $\phi(x)$
- Or prove that $k(x, x')$ corresponds to some $\phi(x)$
  - Never need to actually compute $\phi(x)$

Kernelized SVM looks a lot like an RBF net



Kernel matrix

- The matrix

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \ldots & \vdots \\ k(\mathbf{x}_i, \mathbf{x}_j) \\ \vdots & \ldots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \ldots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

is called the kernel matrix, or the Gram matrix.

- $\mathbf{K}$ is positive semidefinite

Mercer's theorem relates kernel functions and inner product spaces

- Suppose that for all finite sets of points $\{\boldsymbol{x}_p\}_{p=1}^{N}$ and real numbers $\{\boldsymbol{a}\}_{p=1}^{\infty}$

$$\sum_{i,j} a_j a_i k(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq 0$$

- Then $\boldsymbol{K}$ is called a positive semidefinite kernel
- And can be written as

$$k(\boldsymbol{x}, \boldsymbol{x}') = \phi^T(\boldsymbol{x})\phi(\boldsymbol{x}')$$

- For some vector-valued function $\phi(\boldsymbol{x})$

Some popular kernels

- Polynomial kernel, parameters c and p
$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}' + c)^p$$
  - Finite-dimensional $\phi(\boldsymbol{x})$ can be explicitly computed

- Gaussian or RBF kernel, parameter σ
$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2\sigma}\|\boldsymbol{x} - \boldsymbol{x}'\|^2\right)$$
  - Infinite-dimensional $\phi(\boldsymbol{x})$
  - Equivalent to RBF network, but more principled way of finding centers

Some popular kernels

- Hypebolic tangent kernel, parameters β_1 and β_2

$$k(\boldsymbol{x}, \boldsymbol{x}') = \tanh(\beta_1 \boldsymbol{x}^T \boldsymbol{x}' + \beta_2)$$

  - Only positive semidefinite for some values of β_1 and β_2
  - Inspired by neural networks, but more principled way of selecting number of hidden units

- String kernels or other structure kernels
  - Can prove that they are positive definite
  - Computed between non-numeric items
  - Avoid converting to fixed-length feature vectors

Example: polynomial kernel

- Polynomial kernel in 2D, $c = 1, p = 2$

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}' + 1)^2 = (x_1 x_1' + x_2 x_2' + 1)^2$$

$$= x_1^2 {x_1'}^2 + x_2^2 {x_2'}^2 + 2x_1 x_1' x_2 x_2' + 2x_1 x_1' + 2x_2 x_2' + 1$$

- If we define

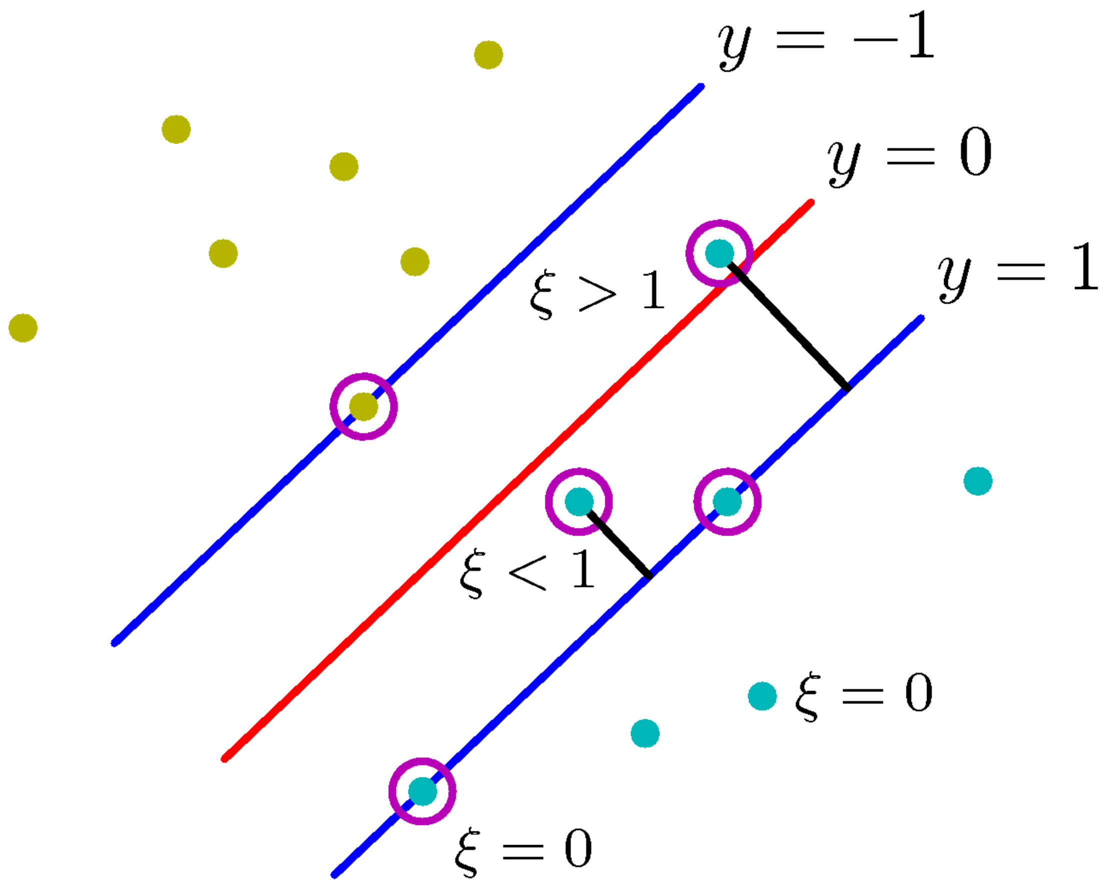
$$\phi(\boldsymbol{x}) = \left[x_1^2, x_2^2, \sqrt{2}\mathrm{x}_1\mathrm{x}_2, \sqrt{2}\mathrm{x}_1, \sqrt{2}\mathrm{x}_2, 1\right]^T$$

- Then $k(\boldsymbol{x}, \boldsymbol{x}') = \phi^T(\boldsymbol{x})\phi(\boldsymbol{x}')$

What if the classes overlap?

- Allow mis-classifications, but penalize them
  - in proportion to distance on the wrong side of the margin
  - Add to existing cost, minimize sum of the two
- Introduce "slack variables" $\xi_p \geq 0$
  - one per training point
  - $\xi_p = \max\left(1 - d_p y\left(\boldsymbol{x}_p\right), 0\right)$
- Interpretation
  - $\xi_p = 0$ for points on the correct side of the margin
  - $0 < \xi_p < 1$ for correctly classified points within margin
  - $\xi_p > 1$ for mis-classified points

Meaning of ξ_p



Incorporate slack variables in optimization

- New problem:

$$\text{argmin}_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_p \xi_p$$

$$\text{Subject to } \; d_p y(\boldsymbol{x}_p) \geq 1 - \xi_p$$

- So constraint $d_p y(\boldsymbol{x}_p) \geq 1$ has been relaxed
- But now minimize the sum of the ξ_ps too
- C controls trade-off between margin and slack
  - As $C \rightarrow \infty$, return to SVM for separable data

Unsupervised learning and Self-organizing maps

Types of learning

- Supervised learning: Detailed desired output is provided externally
- Reinforcement learning: Desired end state of an interaction with environment is provided
  - Learn best actions to take to get there
- Unsupervised learning: Discover structure in data
  - E.g., competitive learning and self organization

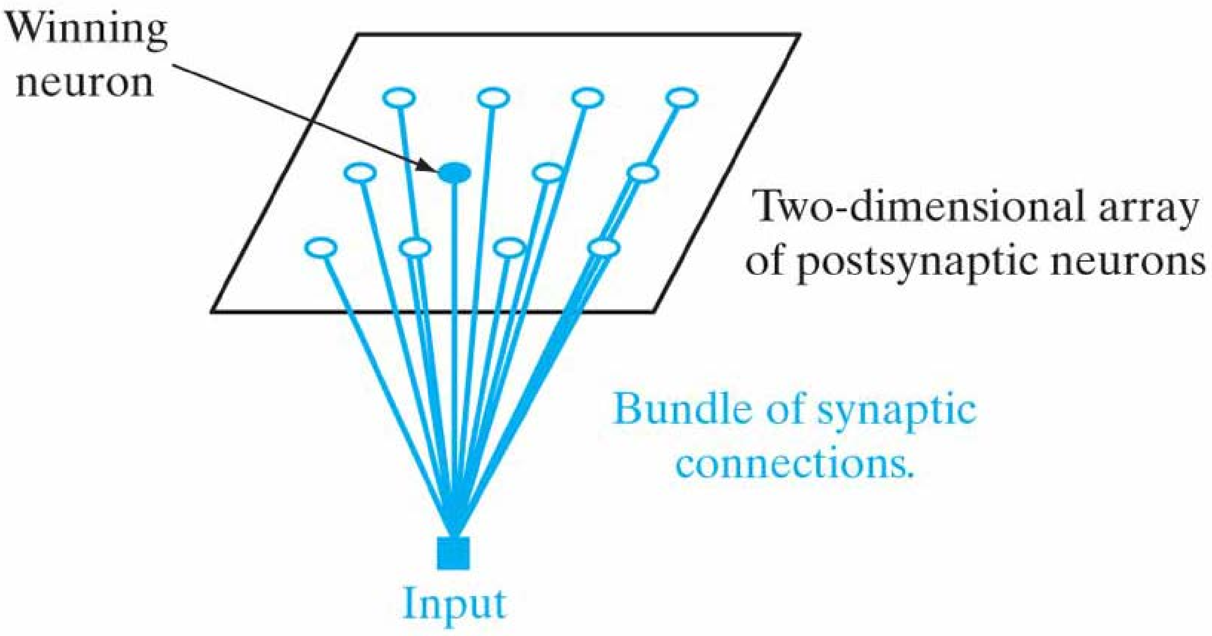
Unsupervised learning

- Goal: learn the distribution of a set of observations
  - Some observations are a better “fit” than others
- Self-organizing maps create spatially coherent internal representations
- Hopfield networks store a set of observations
  - Deterministic, non-linear dynamical system
- Boltzmann machines can behave similarly
  - Stochastic, non-linear dynamical system
- Boltzmann machines with hidden units have a much greater capacity for learning the data distribution

Winner-take-all (WTA) networks implement competitive dynamics

- Recurrent neural network
  - Each neuron excited by input
  - Recurrent dynamics eventually lead to one "winner"
  - Update winning neuron to be more sensitive to that input
- Similar to *K*-means algorithm
- Two different architectures
  - Global inhibition
  - Mutual inhibition

A self-organizing map is a WTA network with a notion of distance between neurons



(b) Kohonen model

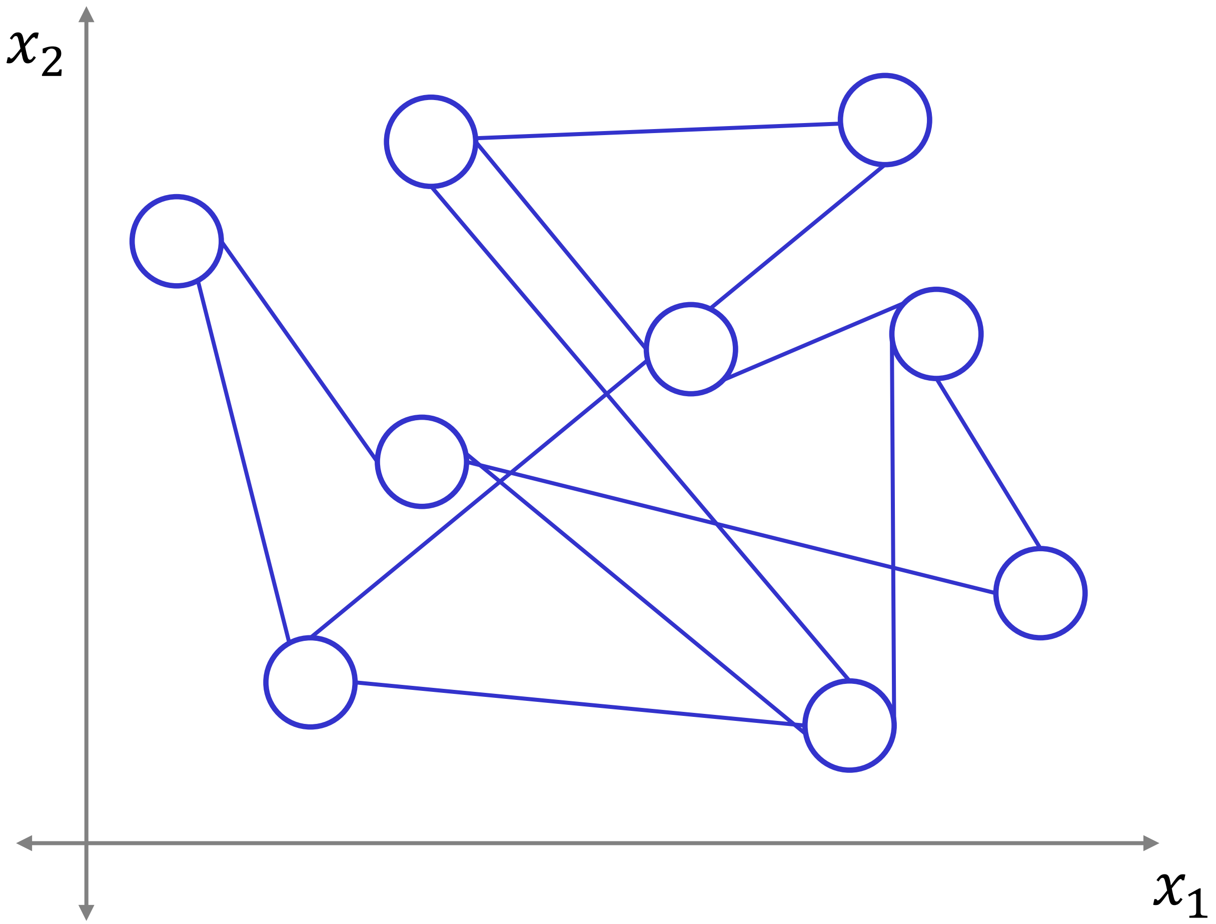
A self-organizing map is a WTA network with a notion of distance between neurons

- Each node in the SOM has a prototype vector
  - Computes activation based on distance to an input
  - What it's looking for or excited by
- Each node in the SOM has a set of neighbors
  - Or a distance function to the rest of the neurons
- Learning in the SOM adjusts the prototypes
  - So that neurons that are "close" to each other have prototypes that are "close" to each other
- Learns a nonlinear dimensionality reduction

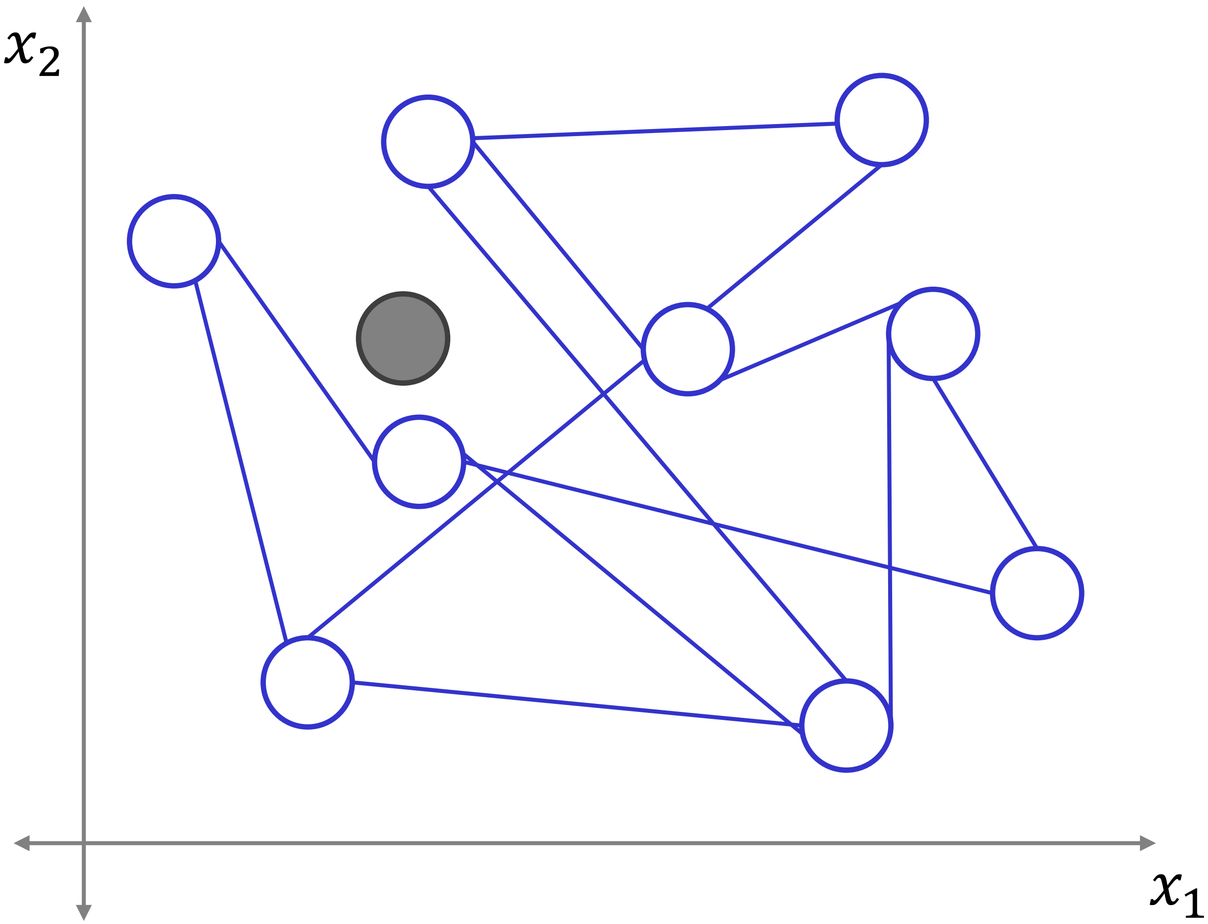
SOM training

- Activate neurons based on distance to inputs
  - Find winner, the neuron most activated
- Update neurons based on distance to winner
  - Winner's prototype is updated to be closer to input
  - Neighbors' prototypes are updated less
  - Far away neurons are not updated
- No global objective being optimized
  - But interesting behavior in practice

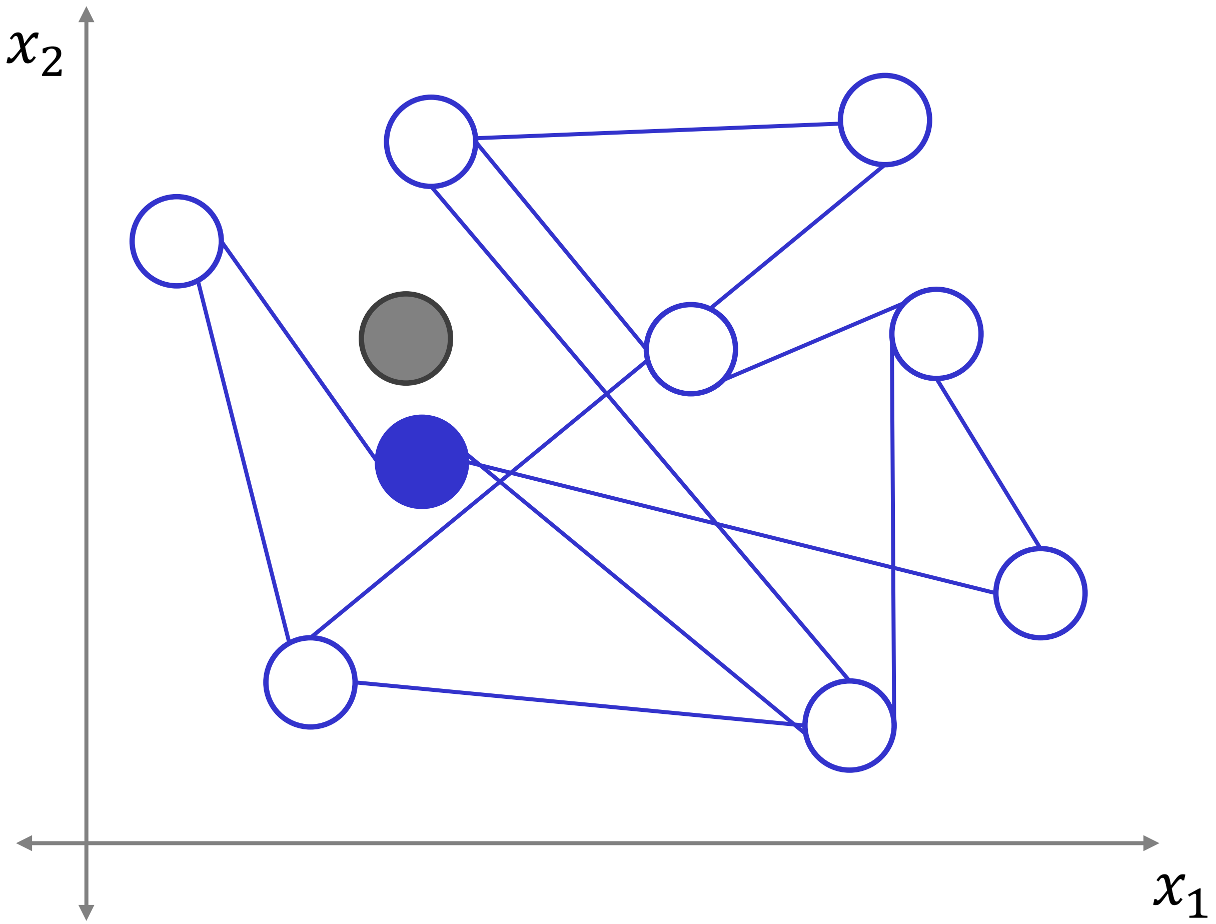
SOM training example: Initial configuration of neuron prototypes



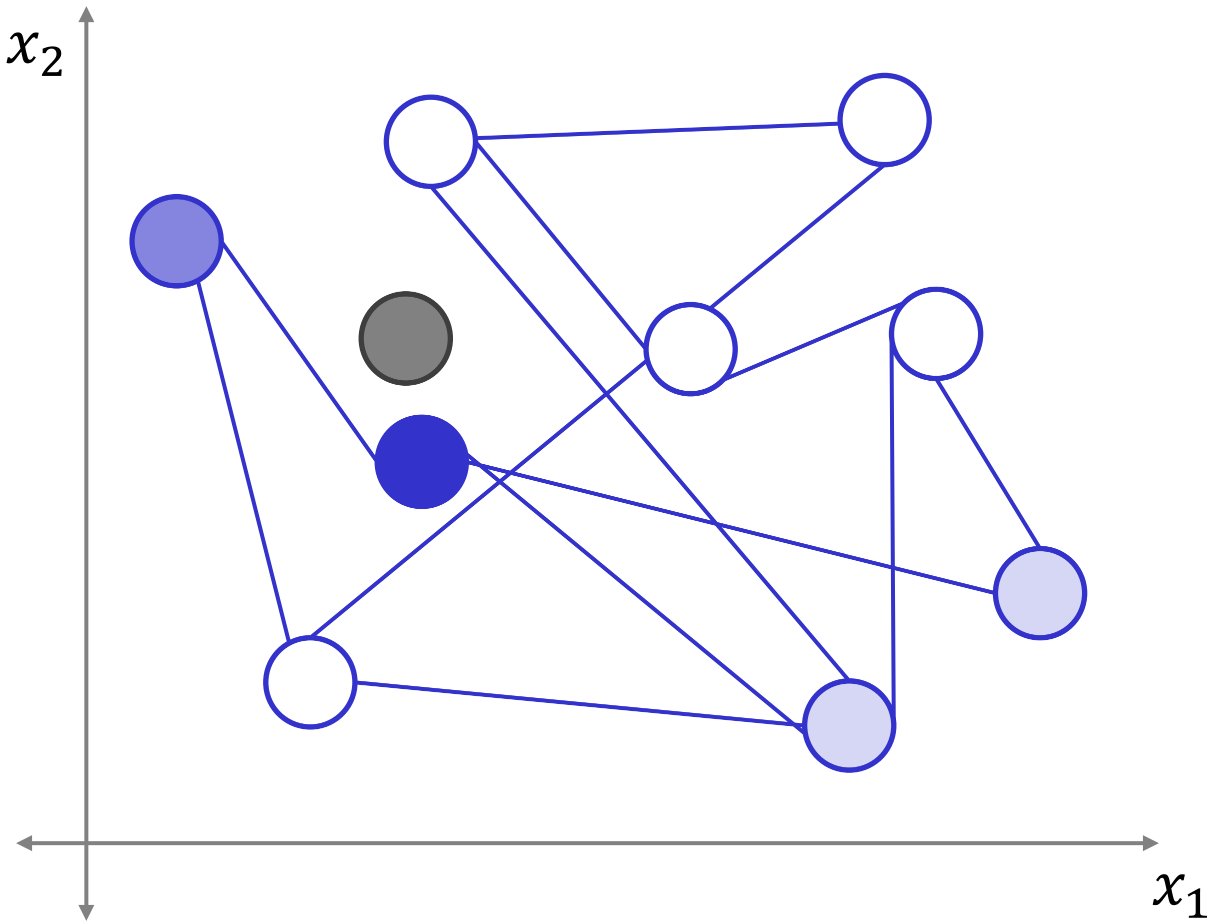
SOM training example: Observe point



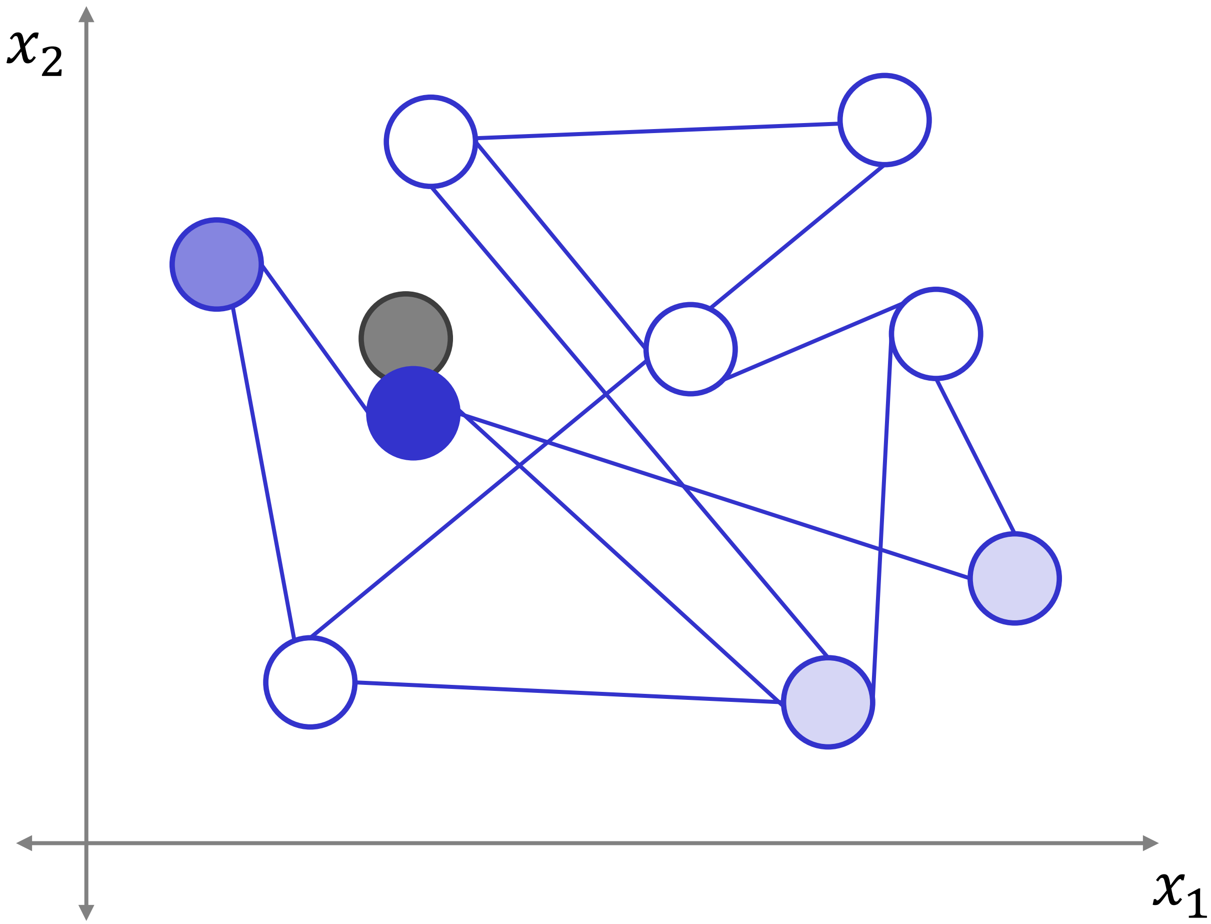
SOM training example: Find closest neuron to observation



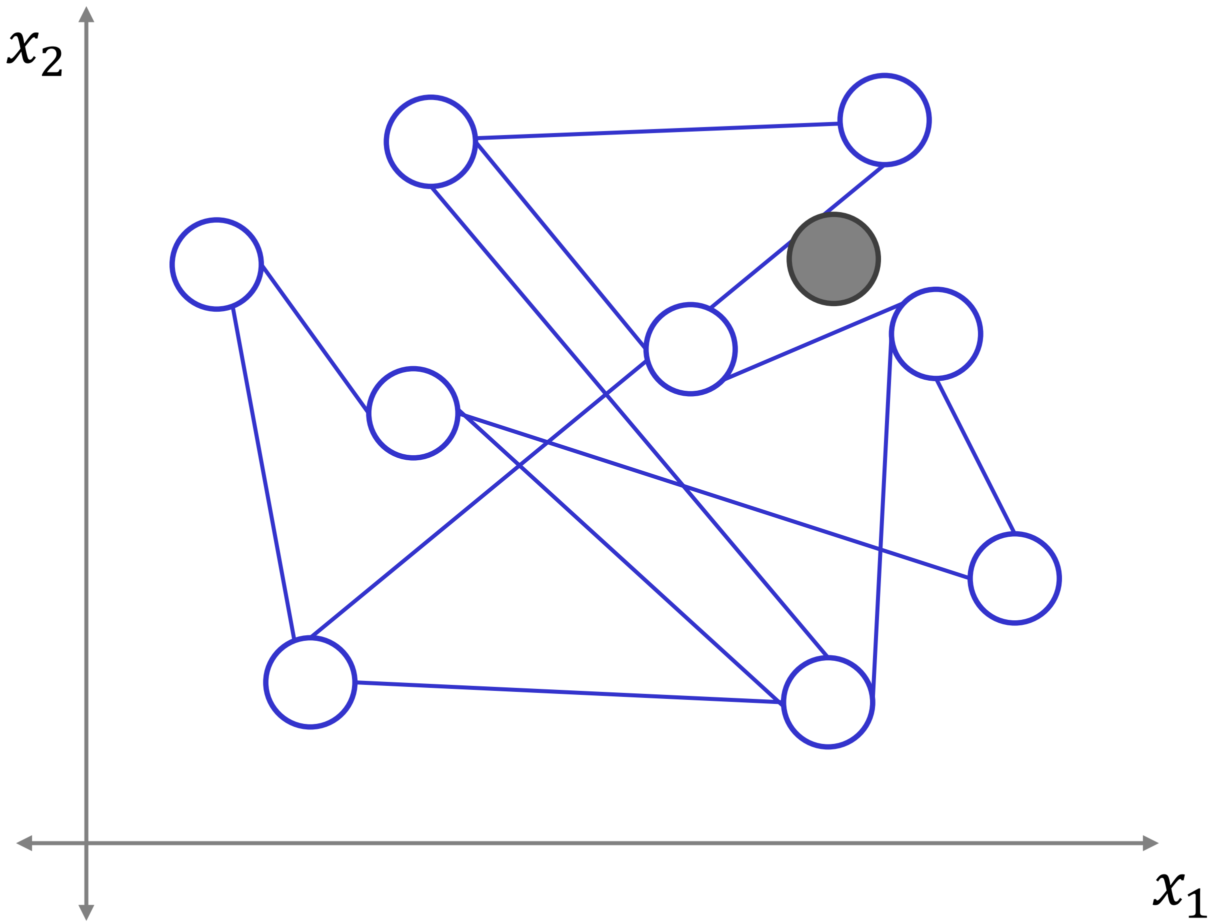
SOM training example:
Activate neurons close **in grid** to that neuron



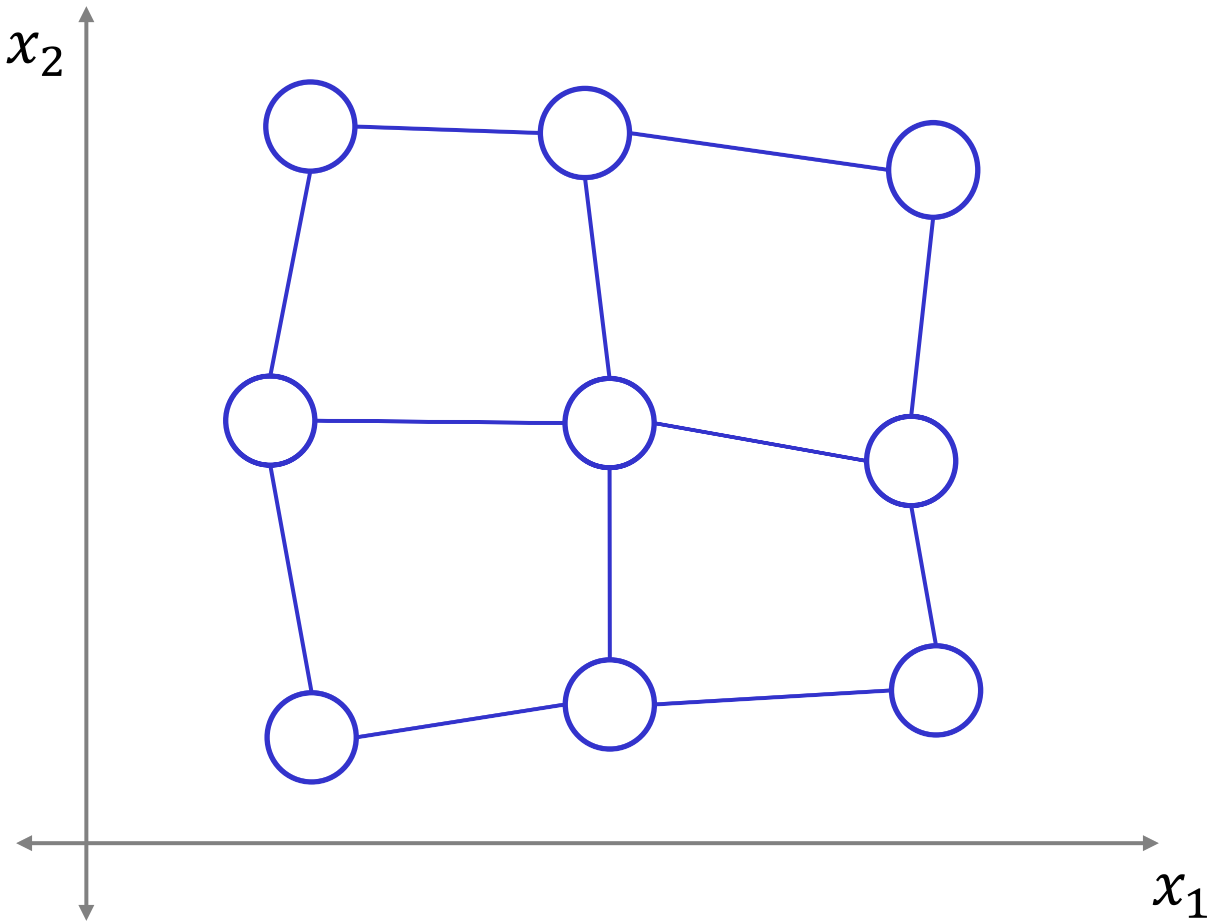
SOM training example:
Move selected neurons towards observation



SOM training example: Observe next point



SOM training example: After many iterations



Hopfield networks

Hopfield networks are unsupervised models that relate new observations to "memories"

- Store a set of "fundamental memories" $\{\boldsymbol{\xi}_1, \boldsymbol{\xi}_2, \dots, \boldsymbol{\xi}_M\}$
- So that when presented with a new pattern $\mathbf{x}$
- The system outputs the stored memory that is most similar to $\mathbf{x}$
- Is that possible to implement as a neural network?
  - Can it be trained to remember any pattern?
  - How many can it store at once?

State of each neuron defines the "state space"

- The network is in state $\boldsymbol{x}_t$ at time t
- The state of the network evolves according to

$$\boldsymbol{x}_{t+1} = \varphi(W\boldsymbol{x}_t + \boldsymbol{b})$$

  - Where we set $\boldsymbol{b} = 0$ without loss of generality
- $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_t\}$ is called a state trajectory
- Goal: set W so that state trajectory of corrupted memory $\boldsymbol{\xi}_i + \Delta$ converges to true memory $\boldsymbol{\xi}_i$

One-shot storage phase uses Hebbian learning

- Hopfield nets set W using the outer-product rule
- For synchronous updates, with N bits

$$W^s = \frac{1}{N}\sum_{\mu=1}^{M} \xi_\mu \xi_\mu^T$$

  - Easier for proving stability of memories
- For asynchronous updates, enforce $W_{ii} = 0$

$$W^a = \frac{1}{N}\sum_{\mu=1}^{M} \xi_\mu \xi_\mu^T - I$$

  - Easier for proving energy minimization

Retrieval phase

- Play out dynamics $\boldsymbol{x}_{t+1} = \varphi(W\boldsymbol{x}_t)$
  - Until reaching a stable state $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t$
  - If argument to $\varphi(\cdot)$ is 0, neuron stays in previous state
    - Leads to symmetric flow diagrams
- Synchronous updates update all bits at once
  - Easier for proving stability of memories
- Asynchronous updates update a random bit at a time
  - Easier for proving energy minimization

Memory capacity for a single bit: Prob of error is defined by amount of cross-talk

- Define, for synchronous updates and W^s

$$C_j^{\vartheta} = -\xi_{\vartheta,j} \sum_i \sum_{\mu \neq \vartheta} \xi_{\mu,j} \xi_{\mu,i} \xi_{\vartheta,i}$$

- Amount cross-talk pushes bit j in the wrong direction

$$C_j^{\vartheta} < 0 \quad \Rightarrow \quad \text{stable}$$
$$0 \leq C_j^{\vartheta} < N \quad \Rightarrow \quad \text{stable}$$
$$C_j^{\vartheta} > N \quad \Rightarrow \quad \text{unstable}$$

Capacity: Lower error prob requires smaller M

P_{error}	M_{max}/N
0.001	0.105
0.0036	0.138
0.01	0.185
0.05	0.37
0.1	0.61

- $P_{\text{error}} = \frac{1}{2}\left(1 - \text{erf}\left(\sqrt{\frac{N}{2M}}\right)\right)$
- So $P_{\text{error}} < 0.01 \Rightarrow M_{\text{max}} = 0.185N$, for one bit
- If we want perfect retrieval for N bits with prob 0.99

$$M_{\text{max}} = \frac{N}{2 \log N}$$

Energy function (Lyapunov function)

- The existence of an energy (Lyapunov) function for a dynamical system ensures its stability
- The energy function for the Hopfield net is

$$E(\mathbf{x}) = -\frac{1}{2}\sum_i \sum_j w_{ji} x_i x_j = -\frac{1}{2}\boldsymbol{x}^T W \boldsymbol{x}$$

- **Theorem**: Given symmetric weights, $w_{ji} = w_{ij}$, the energy function does not increase as the Hopfield net evolves asynchronously

Spurious states

- Not all local minima (stable states) correspond to fundamental memories.
- Other attractors:
  - $-\boldsymbol{\xi}_{\mu}$
  - linear combination of odd number of memories
  - other uncorrelated patterns
- Such attractors are called spurious states

Boltzmann machines

Boltzmann machines are unsupervised probability models

- The primary goal of Boltzmann machine learning is to produce a network that models the probability distribution of observed data (at visible neurons)
  - Such a net can be used for pattern completion, as a part of an associative memory, etc.
- What do we want to do with it?
  - Compute the probability of a new observation
  - Learn parameters of the model from data
  - Estimate likely values completing partial observations

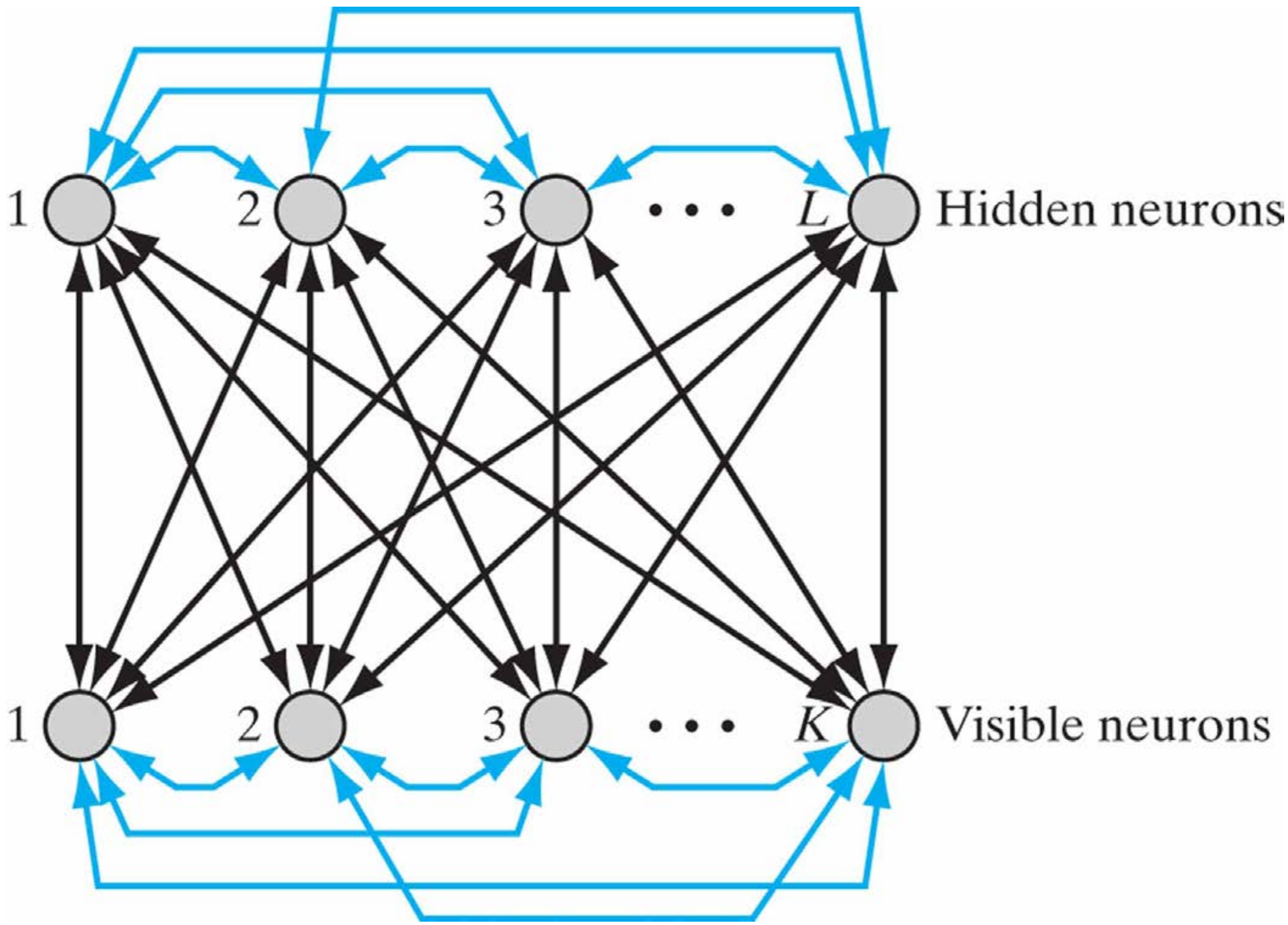
Boltzmann machines have the same energy function as Hopfield networks

- Because of symmetric connections, the energy function of neuron configuration **x** is:

$$E(\mathbf{x}) = -\frac{1}{2}\sum_i \sum_j w_{ji} x_i x_j = -\frac{1}{2}\mathbf{x}^T W \mathbf{x}$$

- Can we start there and derive a probability distribution over configurations?

Boltzmann machines are a stochastic extension of Hopfield networks



The Boltzmann-Gibbs distribution defines probabilities from energies

- Consider a physical system with many states.
  - Let p_i denote the probability of occurrence of state i
  - Let E_i denote the energy of state i
- From statistical mechanics, when the system is in thermal equilibrium, it satisfies

$$p_i = \frac{1}{Z}\exp\left(-\frac{E_i}{T}\right) \quad \text{where} \quad Z = \sum_i \exp\left(-\frac{E_i}{T}\right)$$

  - Z is called the partition function, and T is called the temperature
  - The Boltzmann-Gibbs distribution

Remarks

- Lower energy states have higher probability of occurrences
- As *T* decreases, the probability is concentrated on a small subset of low energy states

Boltzmann-Gibbs distribution applied to Hopfield network energy function

$$p(\boldsymbol{x}) = \frac{1}{Z}\exp\left(-\frac{1}{T}E(\boldsymbol{x})\right) = \frac{1}{Z}\exp\left(\frac{1}{2T}\boldsymbol{x}^T W \boldsymbol{x}\right)$$

- Partition function For N neurons, involves 2^N terms

$$Z = \sum_{\boldsymbol{x}} \exp\left(-\frac{1}{T}E(\boldsymbol{x})\right)$$

- Marginal over H of the neurons Involves 2^H terms

$$p(\boldsymbol{x}_\alpha) = \sum_{\boldsymbol{x}_\beta} p(\boldsymbol{x}_\alpha, \boldsymbol{x}_\beta)$$

Learning can be performed by gradient descent

- The objective of Boltzmann machine learning is to maximize the likelihood of the visible units taking on training patterns by adjusting W
- Assuming that each pattern of the training sample is statistically independent, the log probability of the training sample is:

$$L(\mathbf{w}) = \log \prod_{\mathbf{x}_\alpha} P(\mathbf{x}_\alpha) = \sum_{\mathbf{x}_\alpha} \log P(\mathbf{x}_\alpha)$$

Gradient of log likelihood of data has two terms

$$\frac{\partial L(\mathbf{w})}{\partial w_{ji}} = \frac{1}{T}\left(\rho_{ji}{}^{+} - \rho_{ji}{}^{-}\right)$$

- Where
  $\rho_{ji}{}^{+} = \sum_{\mathbf{x}_\alpha} \sum_{\mathbf{x}_\beta} P\left(\mathbf{x}_\beta \middle| \mathbf{x}_\alpha\right) x_j x_i = \sum_\alpha E_{\boldsymbol{x}_\beta | \boldsymbol{x}_\alpha}\left(x_j x_i\right)$
  - is the mean correlation between neurons *i* and *j* when the visible units are "clamped" to $\mathbf{x}_\alpha$
- And $\rho_{ji}{}^{-} = \sum_{\mathbf{x}} P(\mathbf{x}) x_j x_i = E_{\boldsymbol{x}_\beta, \boldsymbol{x}_\alpha}\left(x_j x_i\right)$
  - is the mean correlation between *i* and *j* when the machine operates without "clamping"

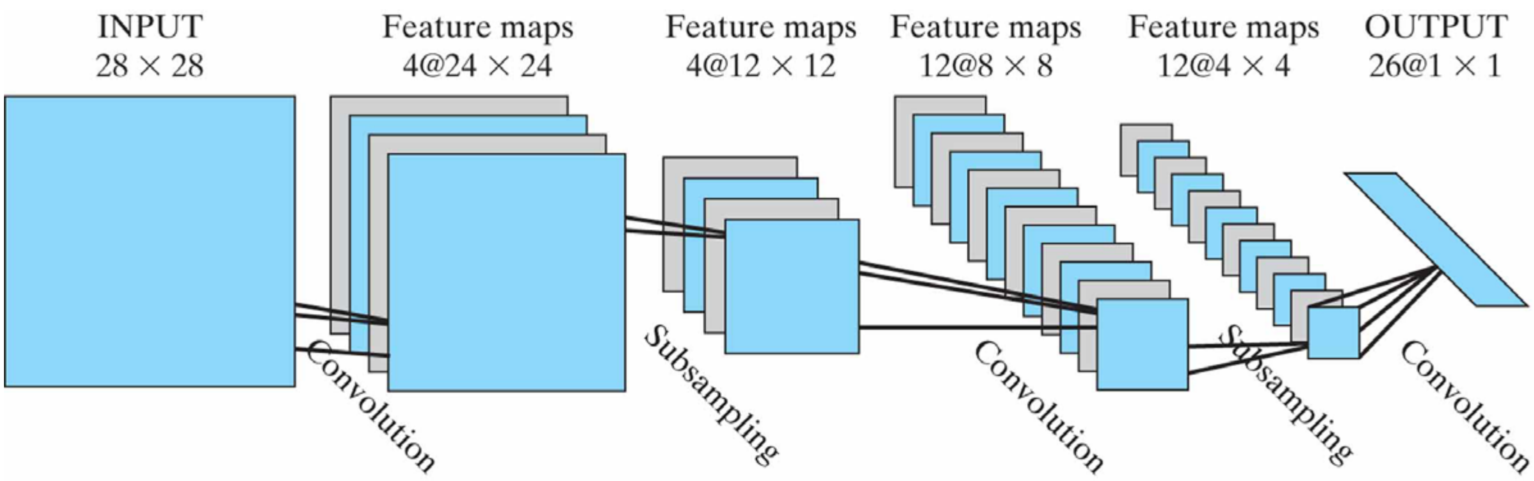
Full Boltzmann machine training algorithm

- The entire algorithm consists of the following nested loops:
  1. Loop over all training data points, accumulating gradient of each weight
  2. For each data point, compute expectation $\langle x_i x_j \rangle$ with $\mathbf{x}_\alpha$ clamped and free
  3. Compute expectations using simulated annealing, gradually decreasing T
  4. For each T, sample the state of the entire net a number of times using Gibbs sampling

Deep Belief Networks

Convolutional networks are deep networks that are feasible to train

- Neural network that learns "receptive fields"
  - And applies them across different spatial positions
- Weight matrices are very constrained
- Train using standard backprop

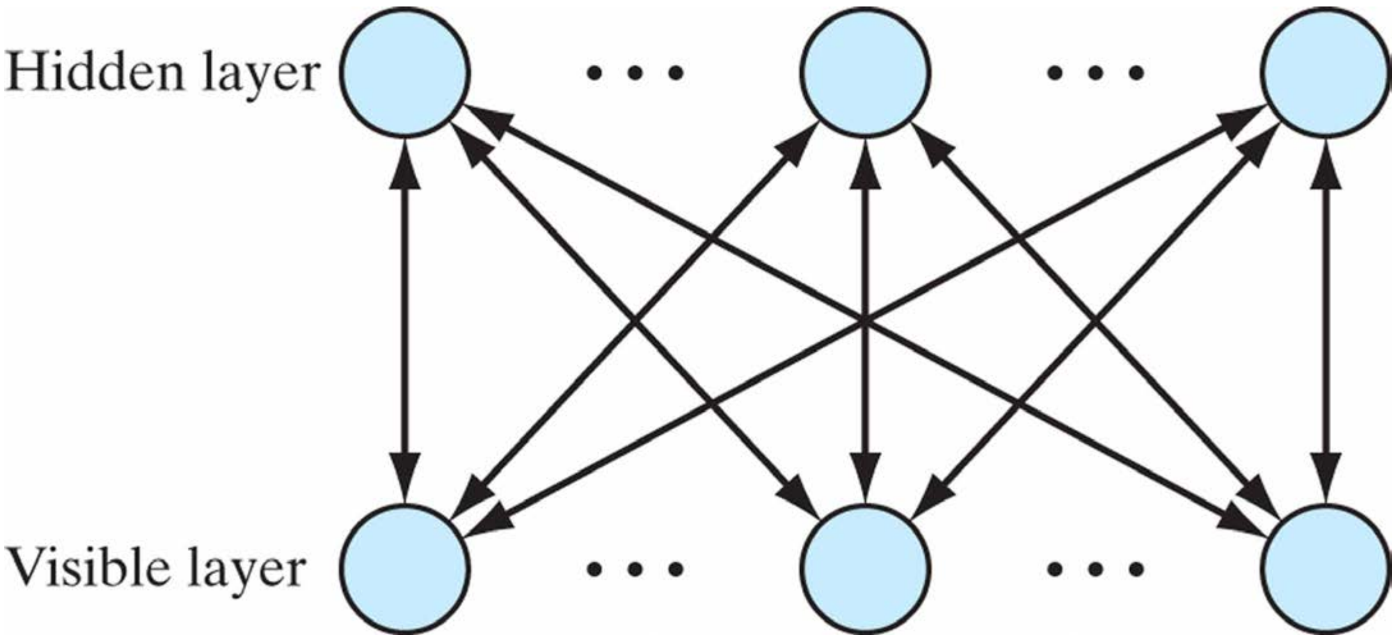


Another way to train deep neural nets is to use unsupervised pre-training

- Build training up from the bottom
  - Train a shallow model to describe the data
  - Treat that as a fixed transformation
  - Train another shallow model on transformed data
  - Etc.
- No long-distance gradients necessary
- Initialize a deep neural network with these params

Restricted Boltzmann machines can be used as building blocks in this way

- A restricted Boltzmann machine (RBM) is a Boltzmann machine with one visible layer and one hidden layer, and no connection within each layer



RBM conditionals are easy to compute

- The energy function is:

$$E(\mathbf{v}, \mathbf{h}) = -\frac{1}{2}\sum_i \sum_j w_{ji} v_j h_i = -\frac{1}{2}\boldsymbol{v}^T W \boldsymbol{h}$$

- So $p(\boldsymbol{v}|\boldsymbol{h}), p(\boldsymbol{h}|\boldsymbol{v})$ are now easy to compute
  - No Gibbs sampling necessary

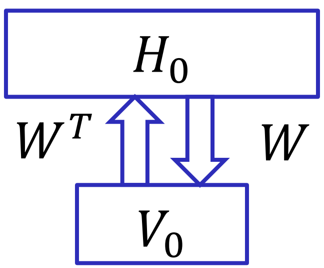
$$p(\boldsymbol{h}|\boldsymbol{v}) = \exp\left(\frac{1}{2}\boldsymbol{v}^T W \boldsymbol{h}\right)\left(\sum_{\boldsymbol{h}} \exp\left(\frac{1}{2}\boldsymbol{v}^T W \boldsymbol{h}\right)\right)^{-1}$$

$$\sum_{\boldsymbol{h}} \exp\left(\frac{1}{2}\boldsymbol{v}^T W \boldsymbol{h}\right) = \prod_i \sum_{h_i} \exp\left(\frac{1}{2}\boldsymbol{v}^T W_{i.} h_i\right)$$

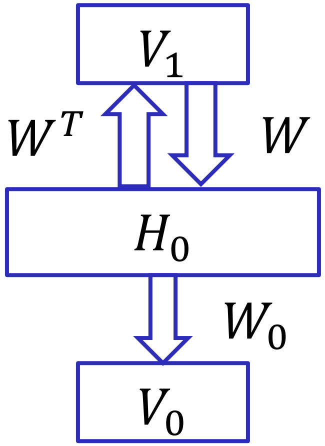
Training a general deep net layer-by-layer

1. First learn W with all weights tied
2. Freeze (fix) W as W^0, which represents the learned weights for the first hidden layer
3. Learn the weights for the second hidden layer by treating responses of the first hidden layer to the training data as “input data”
4. Freeze the weights for the second hidden layer
5. Repeat steps 3-4 as many times as the prescribed number of hidden layers

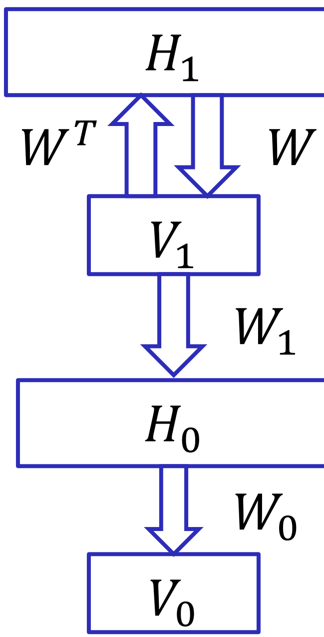
Thus an infinite belief network can be implemented with finite computation



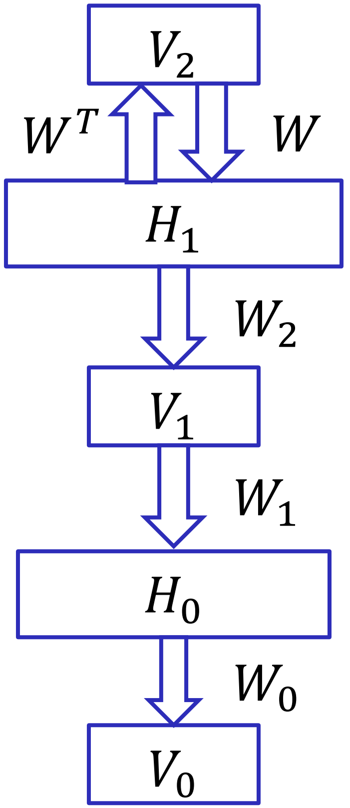
Thus an infinite belief network can be implemented with finite computation



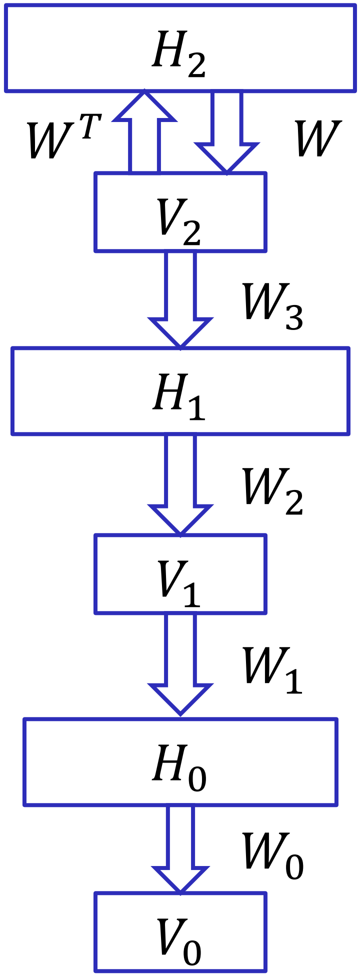
Thus an infinite belief network can be implemented with finite computation



Thus an infinite belief network can be implemented with finite computation



Thus an infinite belief network can be implemented with finite computation



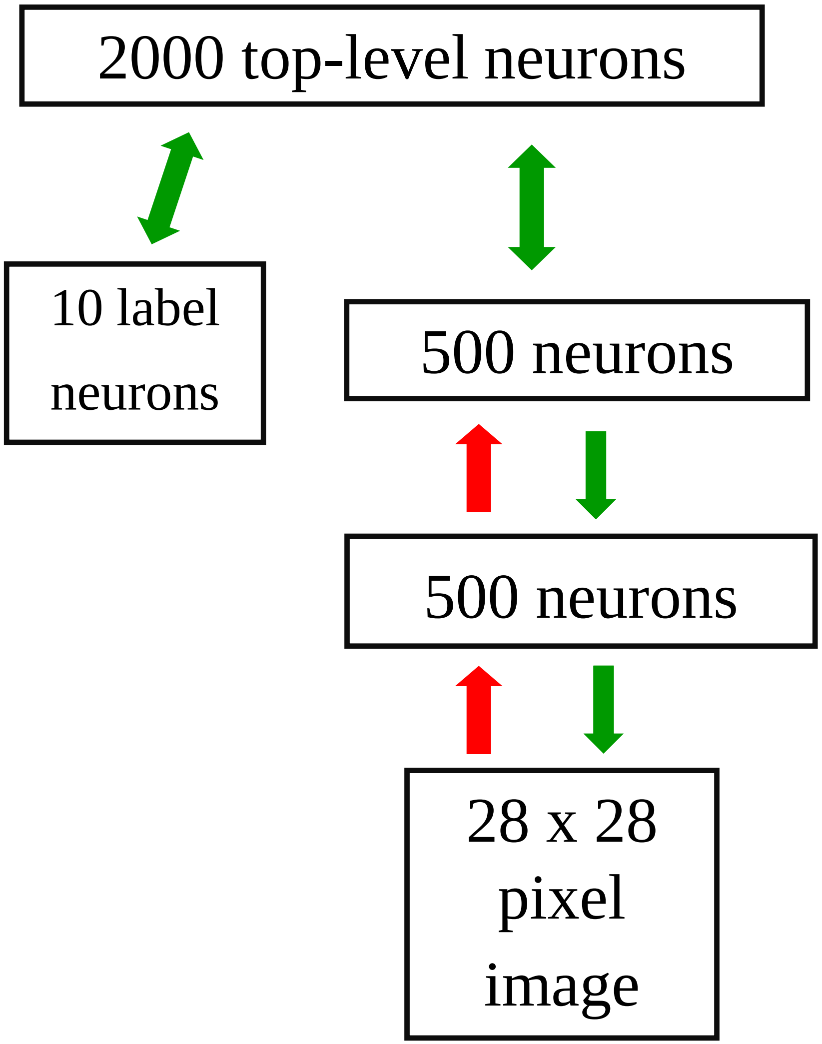
Remarks (Hinton, Osindero, Yeh, 2006)

- As the number of layers increases, the maximum likelihood approximation of the training data improves
- For discriminative training (e.g. for classification) we add an output layer on top of the learned generative model, and train the entire net by a discriminative algorithm
- Although much faster than Boltzmann machines (e.g. no simulated annealing), pretraining is still quite slow, and involves a lot of design as for MLP

DBNs have been successfully applied to an increasing number of tasks

- Ex: MNIST handwritten digit recognition
- A DBN with two hidden layers achieves 1.25% error rate, vs. 1.4% for SVM and 1.5% for MLP
  - DBN with "gentle" discriminative fine-tuning, 1.15%
- Great example animations
  - http://www.cs.toronto.edu/~hinton/digits.html

A neural model of digit recognition



Slide from Hinton MSR Talk

Deep Neural Networks

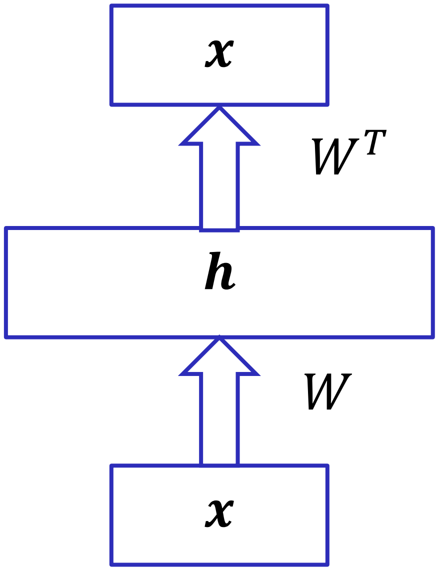
From DBNs to DNNs

- Last lecture described Deep Belief Networks (DBN)
  - Unsupervised, generative, deep models of data
- In practice, DBNs are most useful as initialization for Deep Neural Networks (DNN)
  - Supervised, discriminative, deep function approximators
  - Both have the same structure
  - Weights can be transferred directly, with care
- This lecture covers some DNN details / tricks and autoencoders, another useful unsupervised approach

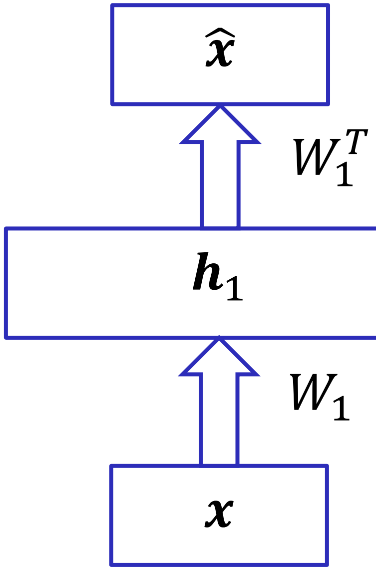
Autoencoders are unsupervised, deterministic networks

- Function $f(\boldsymbol{x})$ trained to predict $\boldsymbol{x}$
  - Can be a standard MLP: $f(\boldsymbol{x}) = \varphi(W_1 \varphi(W_0 x))$
  - Typically, parameters are "tied" so $W_1 = W_0^T$
- Data $\boldsymbol{x}$ provides its own "supervision" signal
- Some limitation prevents the network from learning the identity function
  - Hidden state of smaller dimension than $\boldsymbol{x}$
  - Noisy input (denoising autoencoder)
  - Penalize uninteresting solutions (contractive autoencoder)
  - Etc.

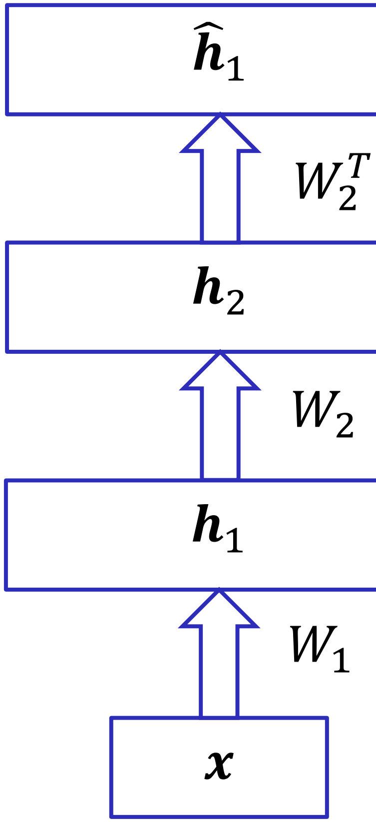
Autoencoder architecture



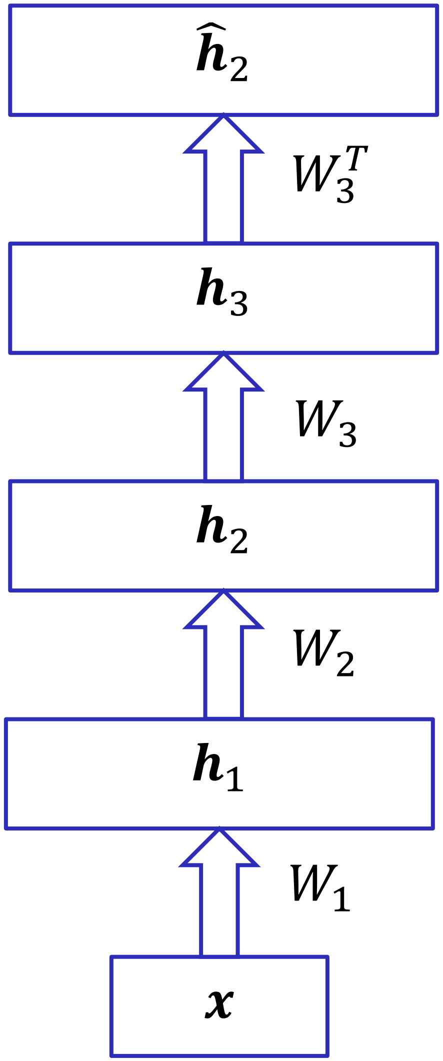
Autoencoders can also be stacked to initialize a deep neural network



Autoencoders can also be stacked to initialize a deep neural network



Autoencoders can also be stacked to initialize a deep neural network



Each type of data leads to a particular error function and a particular output unit type

- Constraints lead to error functions
  - From negative log likelihood of distributions
- Error functions lead to output non-linearities
  - That put the gradients in a particularly nice form

$$\nabla E(\boldsymbol{w}) = -\sum_{p,k} (d_k - y_{pk}) \boldsymbol{a}_p$$

- In general: distributions in the exponential family work nicely with output units in the form of the "canonical link function"

Each type of data leads to a particular error function and a particular output unit type

Data type	Error function $E(\boldsymbol{w})$	Output unit y_k
Unconstrained	$\frac{1}{2}\sum_{p,k}(d_{pk} - y_k)^2$	$\boldsymbol{w}_k^T\boldsymbol{a}$
Binary (Bernoulli)	$-\sum_{p,k} d_k \log y_{pk} + (1 - d_k)\log(1 - y_{pk})$	$\frac{1}{1 + \exp(-\boldsymbol{w}_k^T\boldsymbol{a})}$
Multinomial	$-\sum_{p,k} d_k \log y_{pk}$	$\frac{\exp(-\boldsymbol{w}_k^T\boldsymbol{a})}{\sum_{k'} \exp(-\boldsymbol{w}_{k'}^T\boldsymbol{a})}$