

Neural Networks

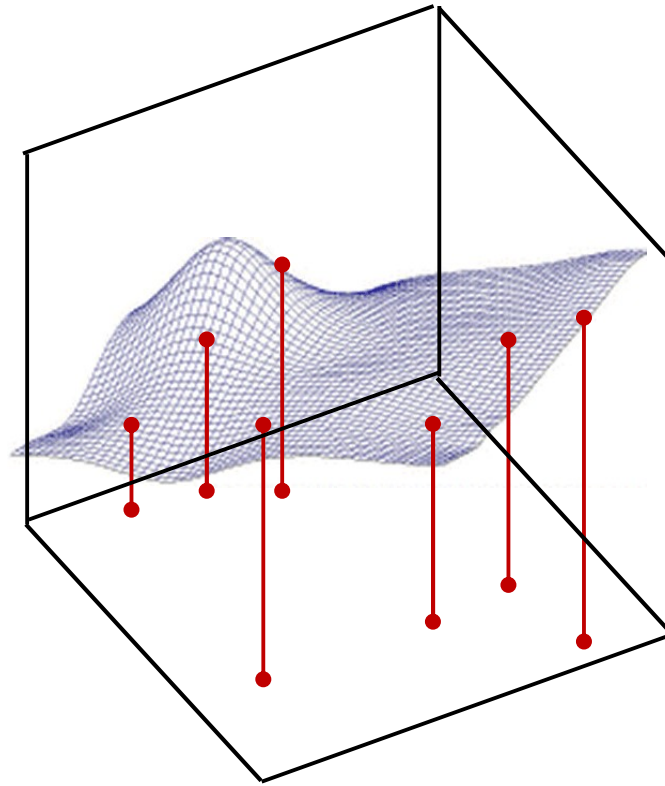
Representations

Spring 2021

Story so far

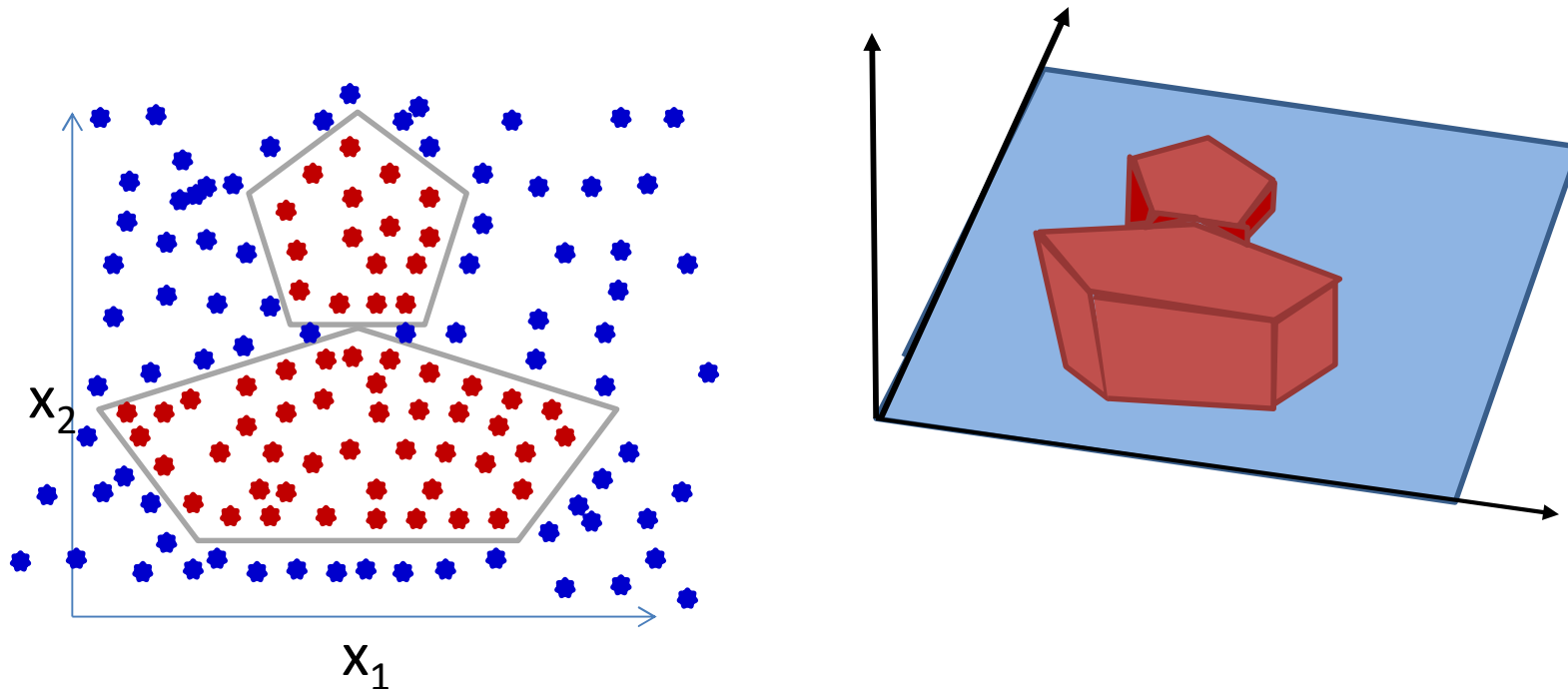
- Neural nets are universal approximators
 - They can model any Boolean, categorical or real-valued function
- They can verify static inputs for patterns
- They can scan for patterns
- They can analyze time series for patterns
- They must be *trained* to make their predictions
- But what do they learn *internally*?
 - What does the network actually represent?

Learning in the net



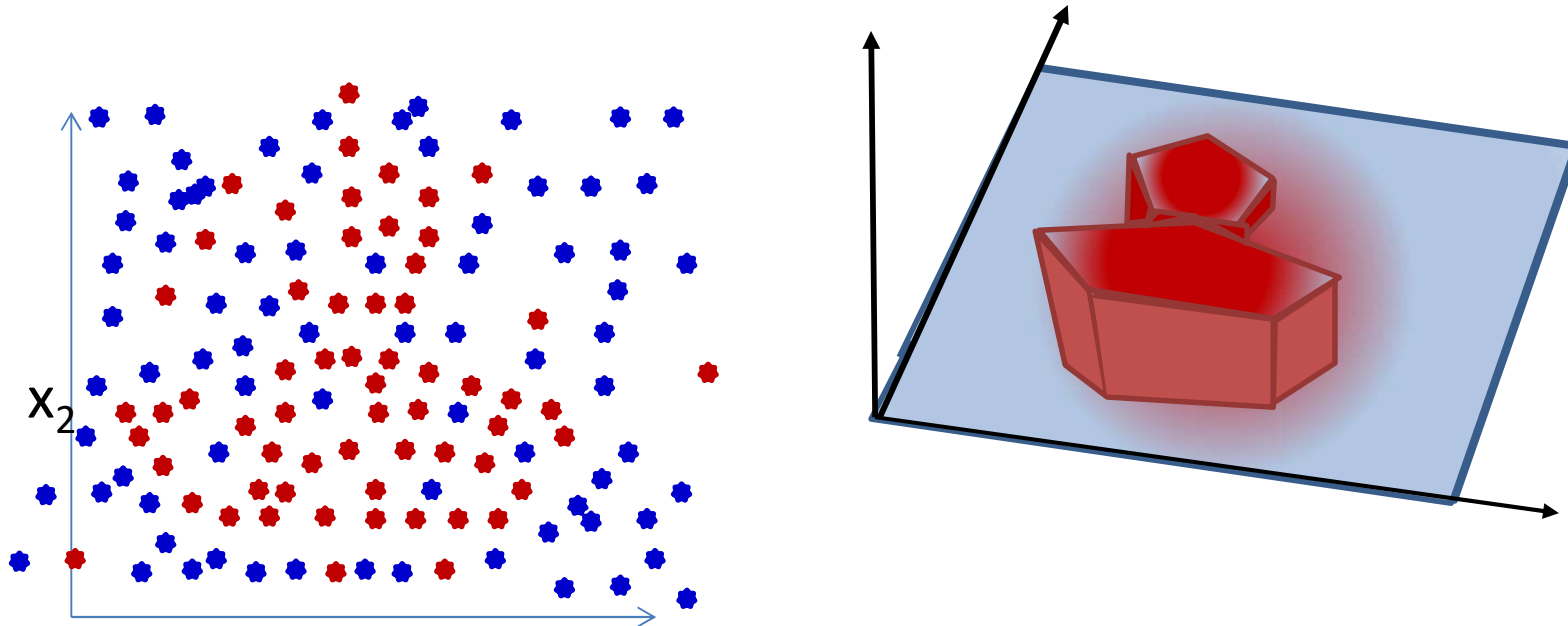
- Problem: Given a collection of input-output pairs, learn the function

Learning for classification



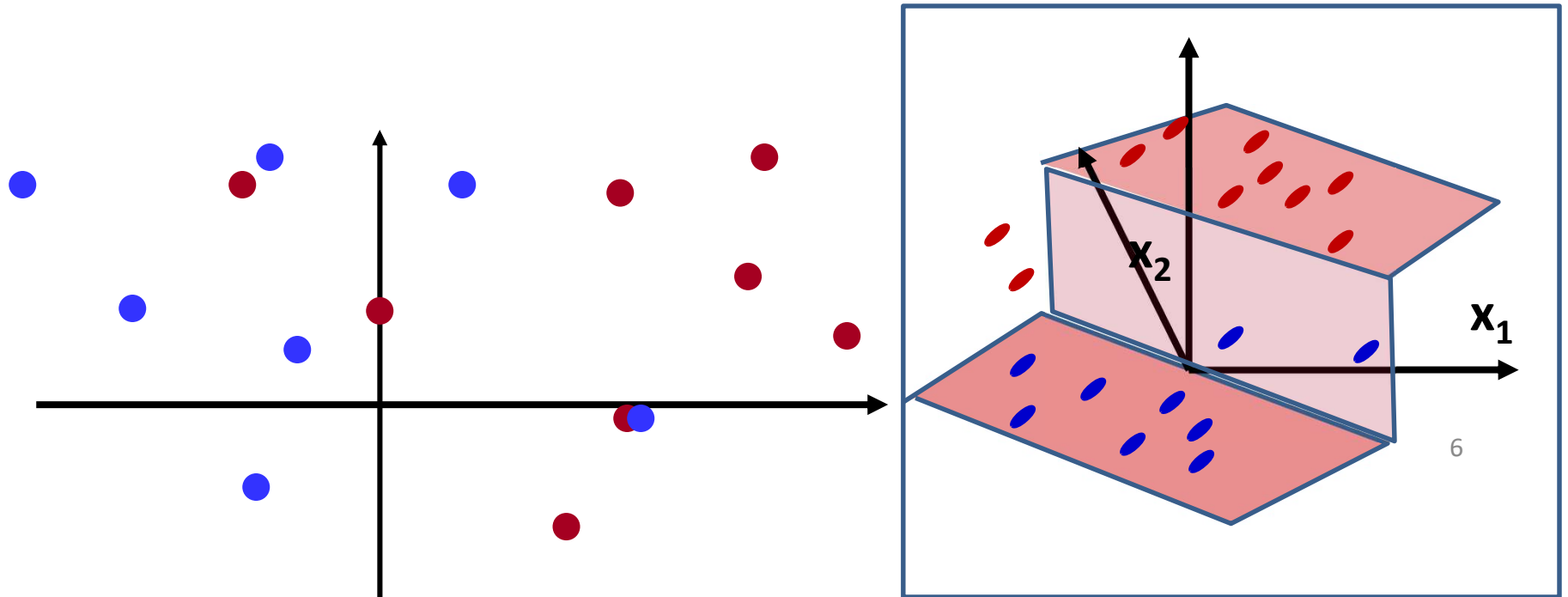
- When the net must learn to classify..
 - Learn the classification boundaries that separate the training instances

Learning for classification



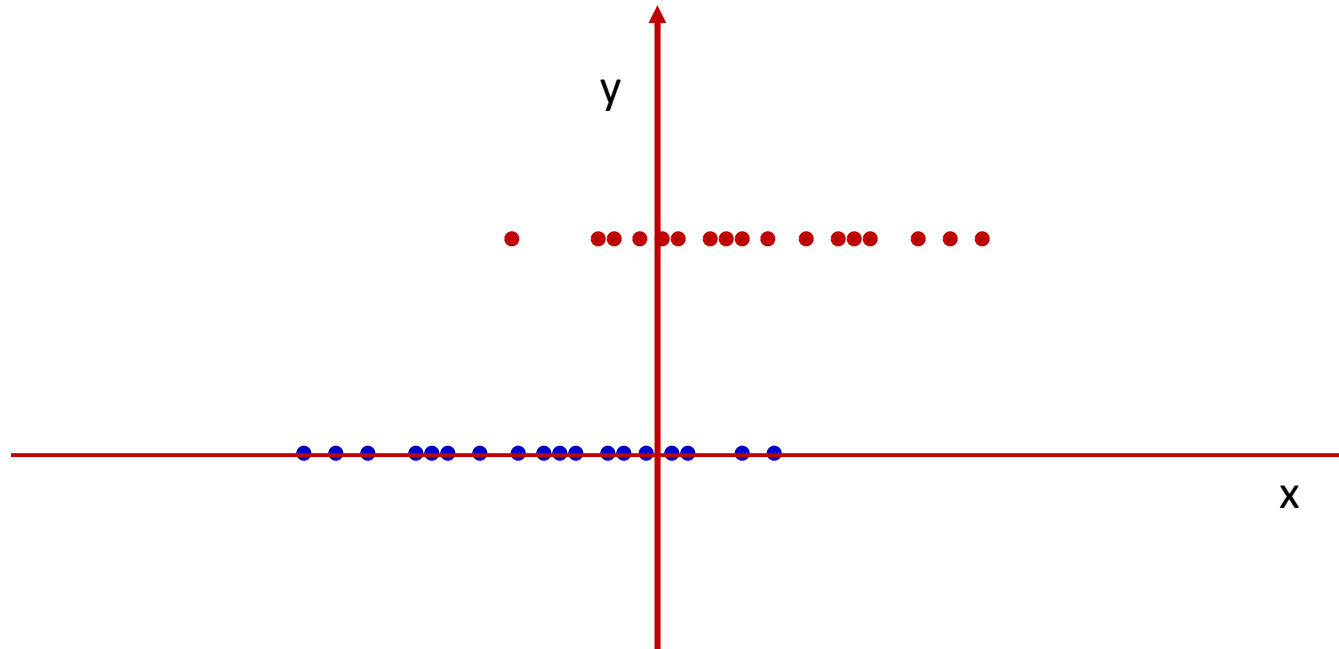
- In reality
 - In general not really cleanly separated
 - So what is the function we learn?

In reality: Trivial linear example



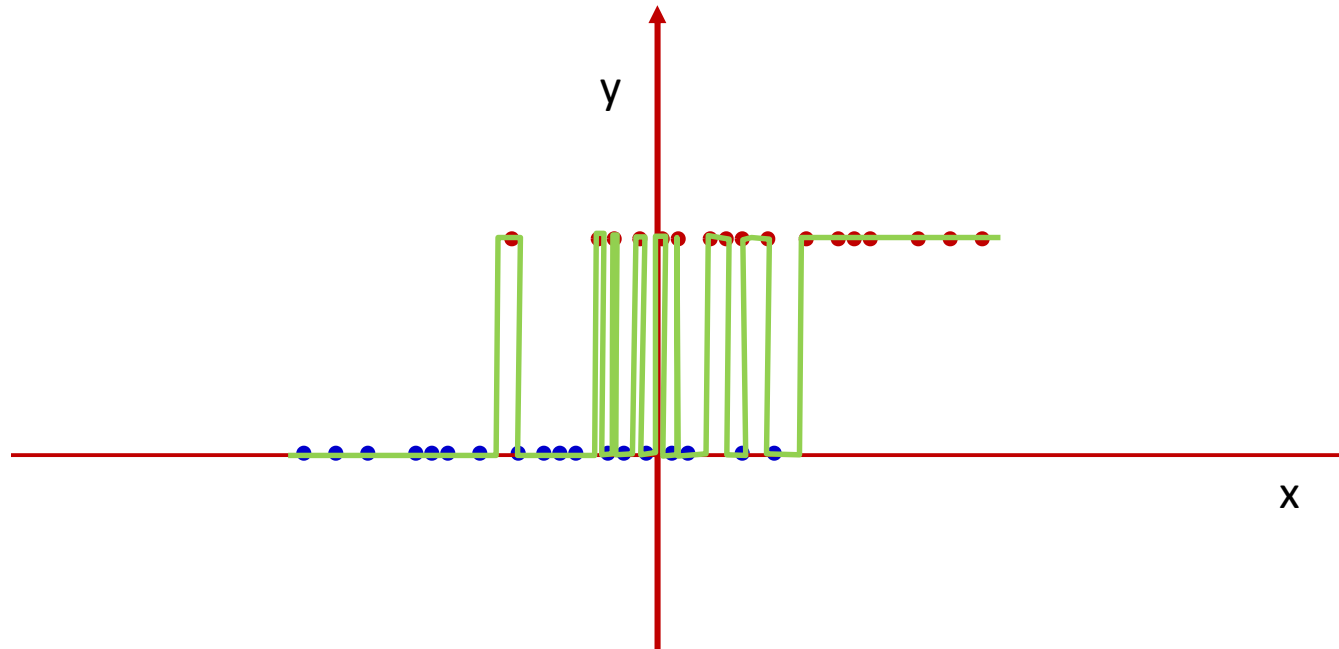
- Two-dimensional example
 - Blue dots (on the floor) on the “red” side
 - Red dots (suspended at $Y=1$) on the “blue” side
 - No line will cleanly separate the two colors

Non-linearly separable data: 1-D example



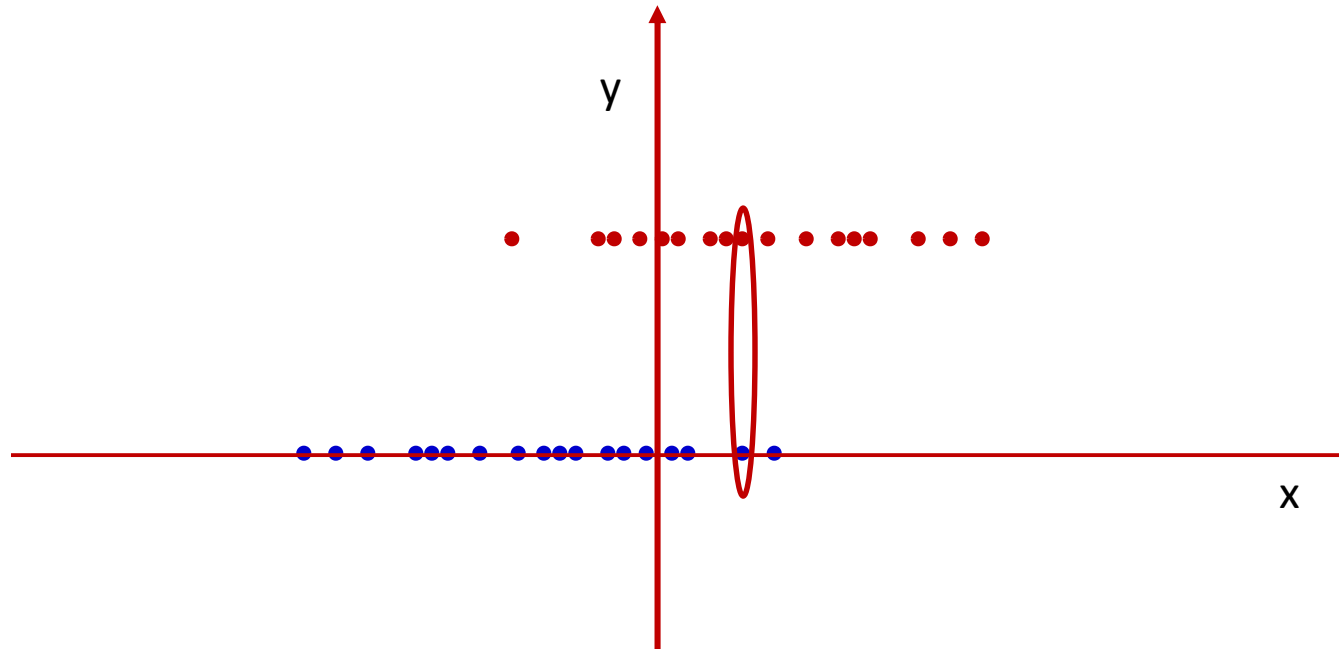
- One-dimensional example for visualization
 - All (red) dots at $Y=1$ represent instances of class $Y=1$
 - All (blue) dots at $Y=0$ are from class $Y=0$
 - The data are not linearly separable
 - In this 1-D example, a linear separator is a threshold
 - No threshold will cleanly separate red and blue dots

Undesired Function



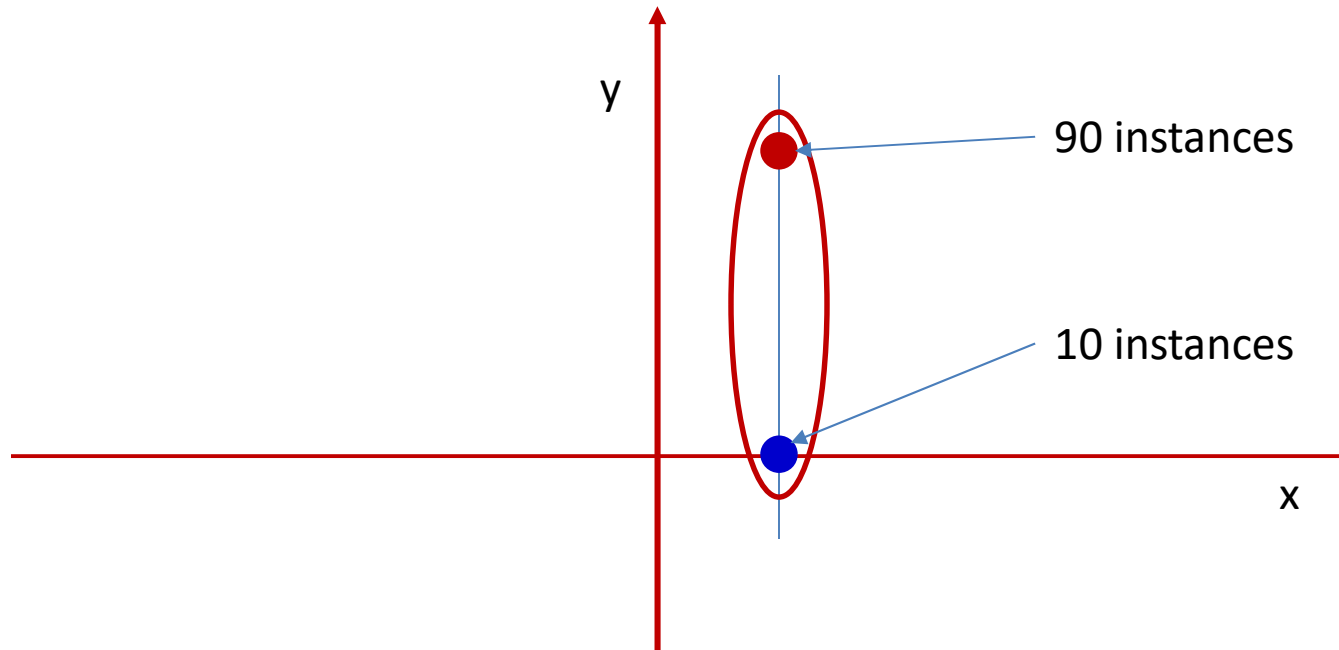
- One-dimensional example for visualization
 - All (red) dots at $Y=1$ represent instances of class $Y=1$
 - All (blue) dots at $Y=0$ are from class $Y=0$
 - The data are not linearly separable
 - In this 1-D example, a linear separator is a threshold
 - No threshold will cleanly separate red and blue dots

What if?



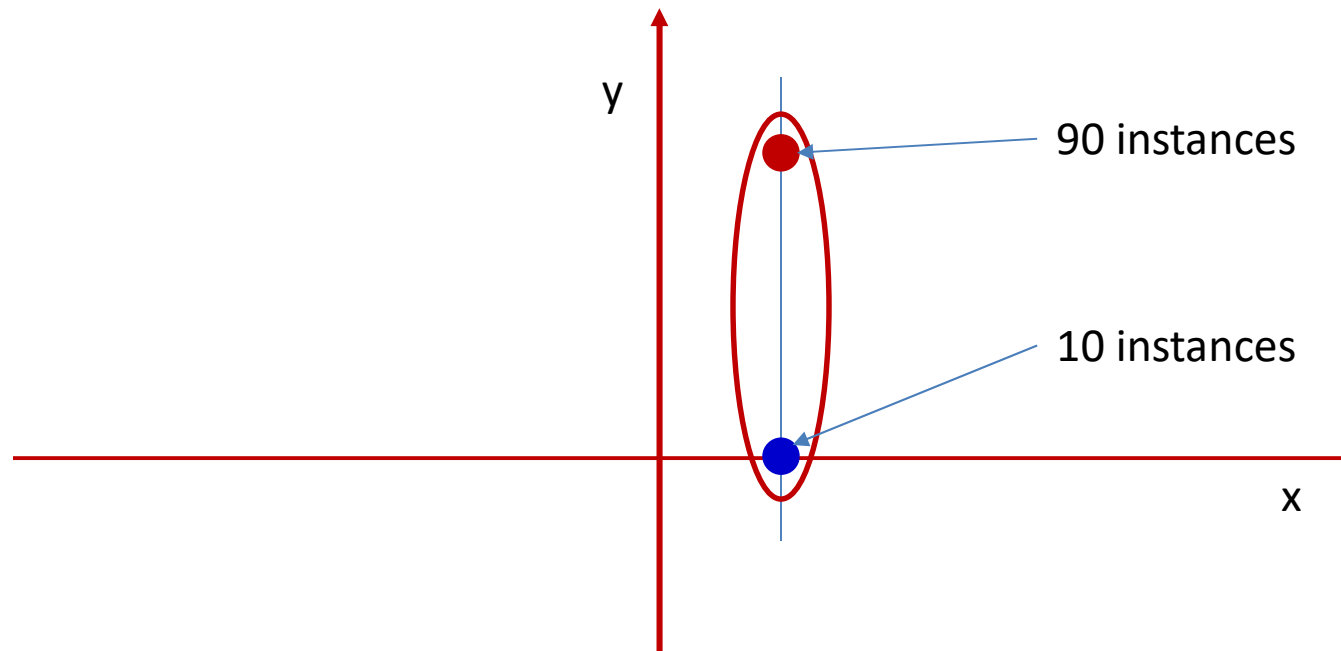
- One-dimensional example for visualization
 - All (red) dots at $Y=1$ represent instances of class $Y=1$
 - All (blue) dots at $Y=0$ are from class $Y=0$
 - The data are not linearly separable
 - In this 1-D example, a linear separator is a threshold
 - No threshold will cleanly separate red and blue dots

What if?



- What must the value of the function be at this x ?
 - 1 because red dominates?
 - 0.9 : The average?

What if?



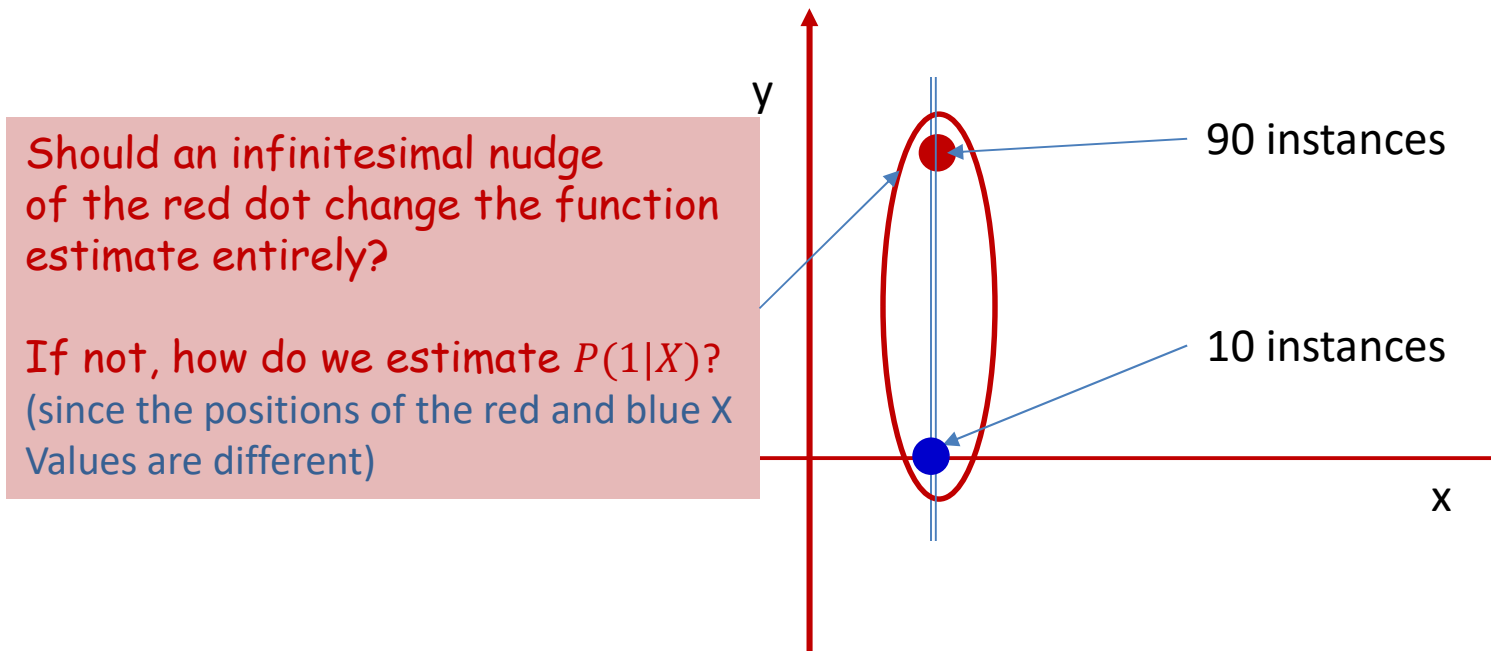
- What must the value of the function be at this X ?

- 1 because red dominates?
- 0.9 : The average?

Estimate: $\approx P(1|X)$

Potentially much more useful than a simple 1/0 decision
Also, potentially more realistic

What if?



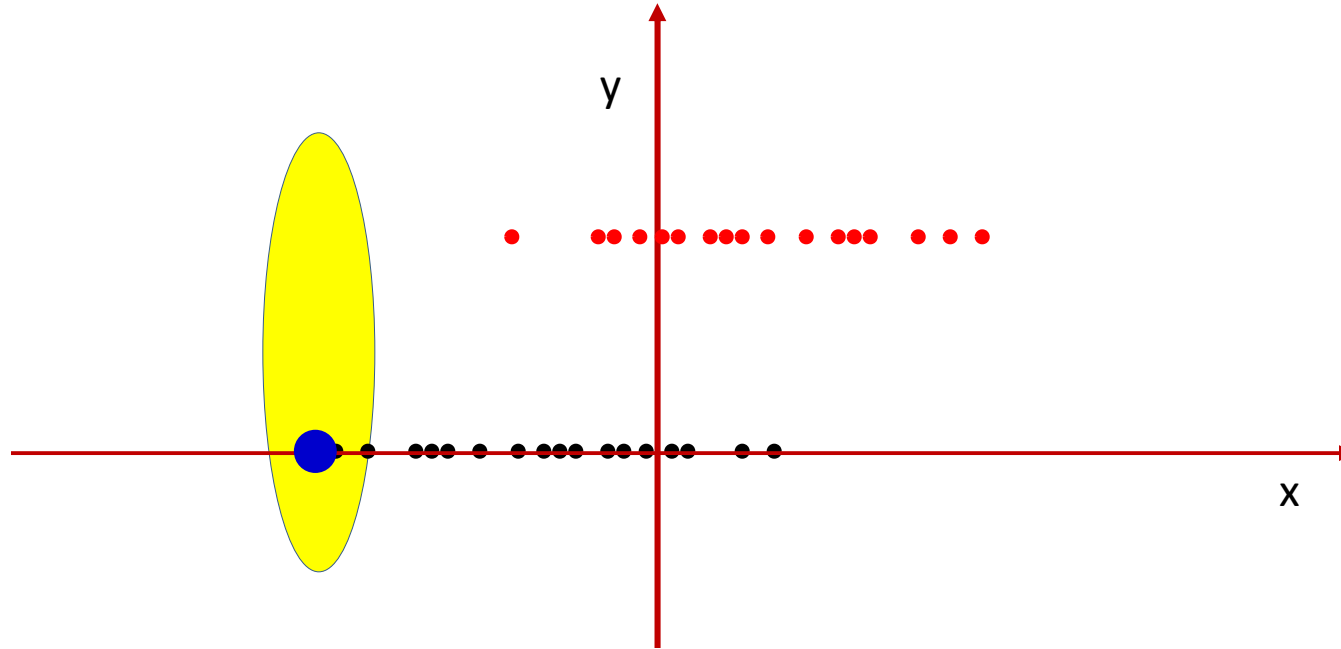
- What must the value of the function be at this X ?

- 1 because red dominates?
- 0.9 : The average?

Estimate: $\approx P(1|X)$

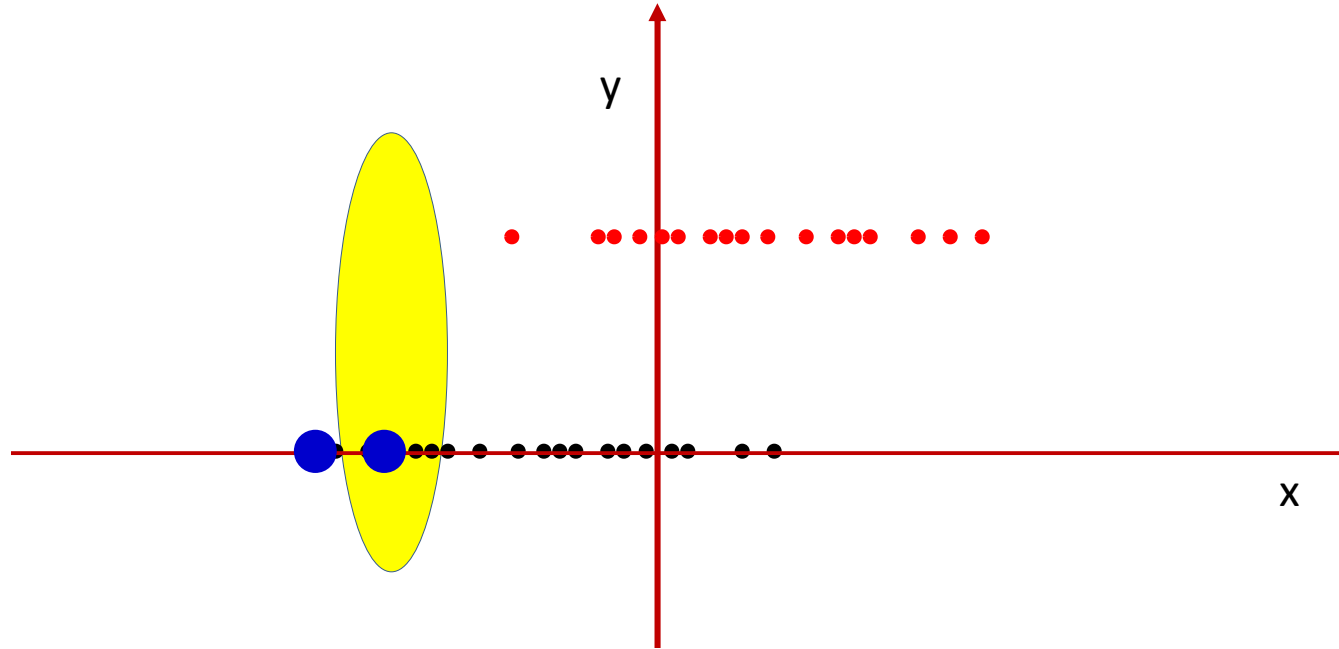
Potentially much more useful than a simple 1/0 decision
Also, potentially more realistic

The *probability* of $y=1$



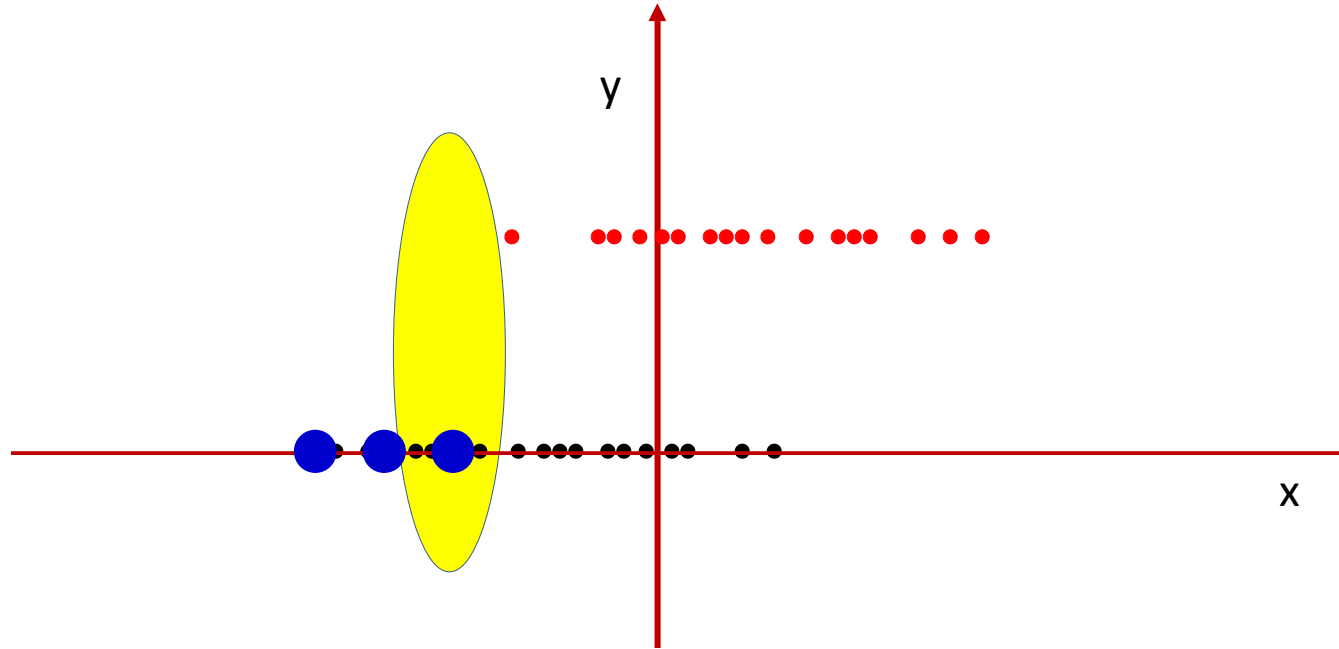
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of $Y=1$ at that point

The *probability* of $y=1$



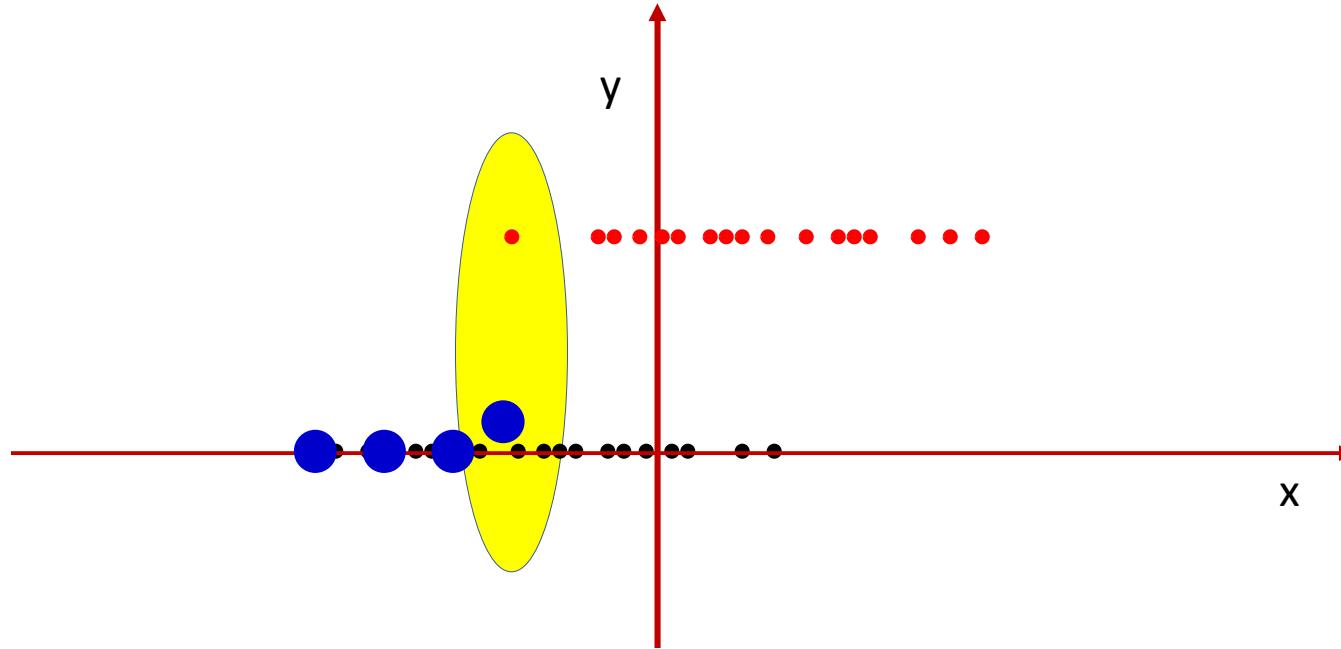
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



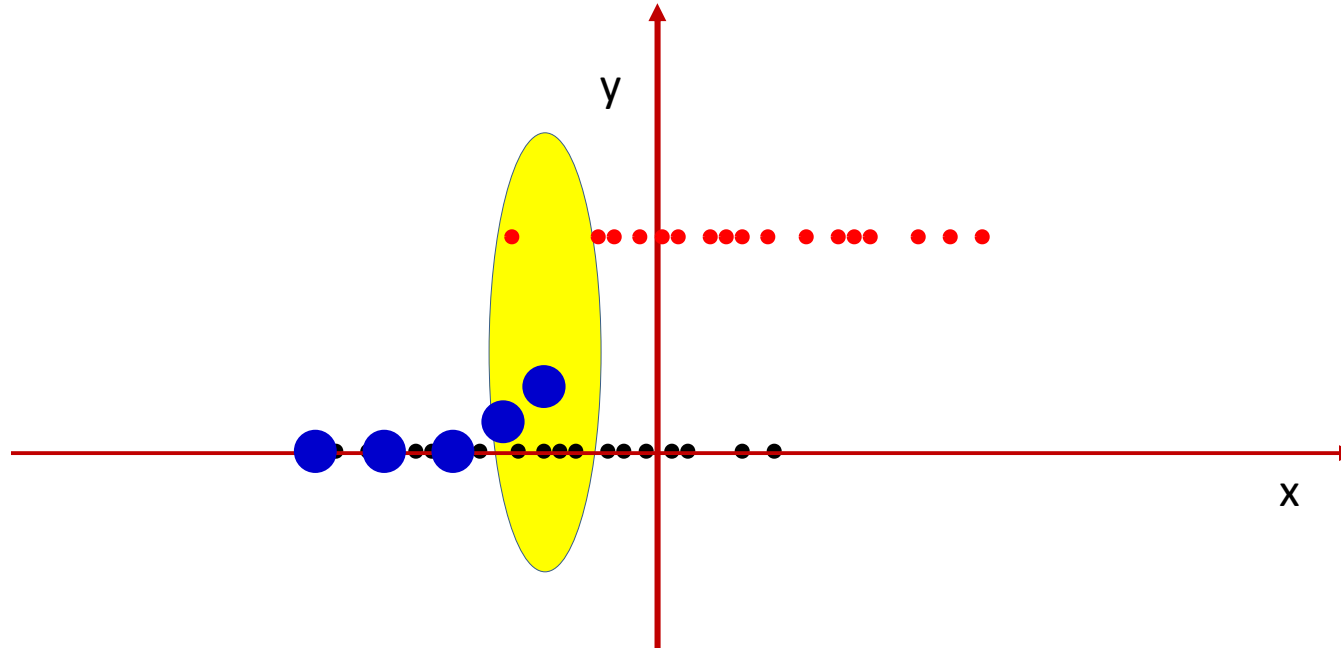
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



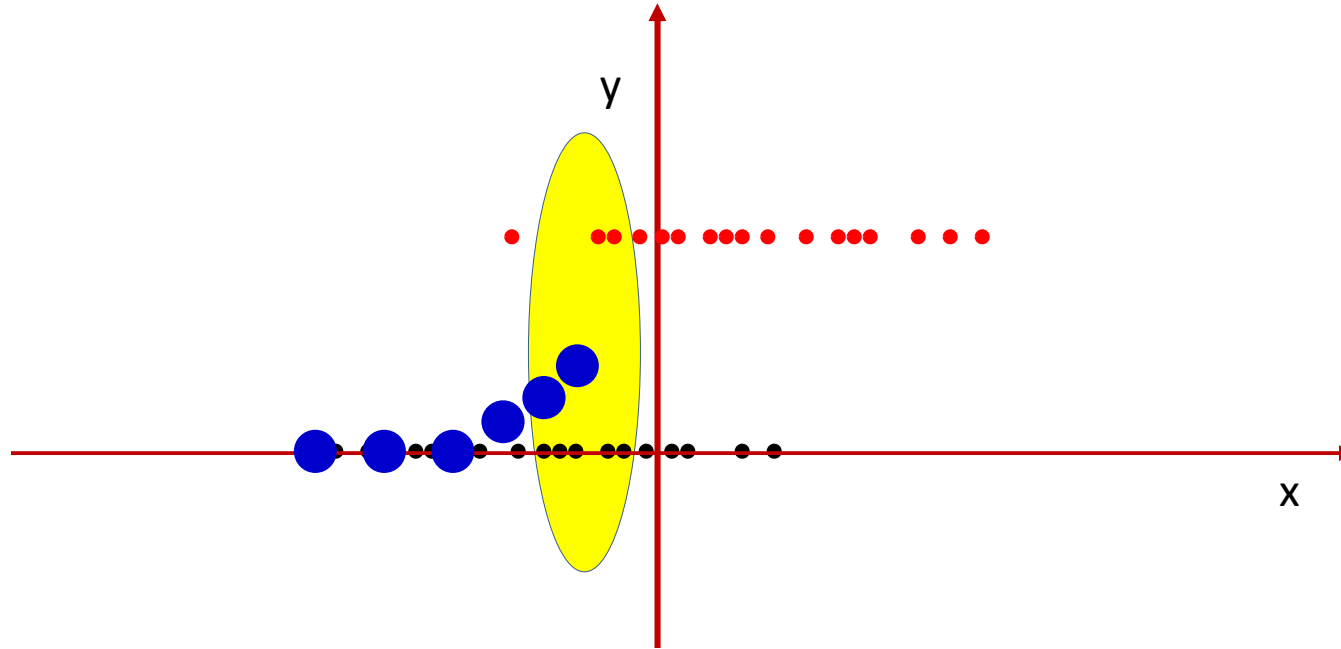
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



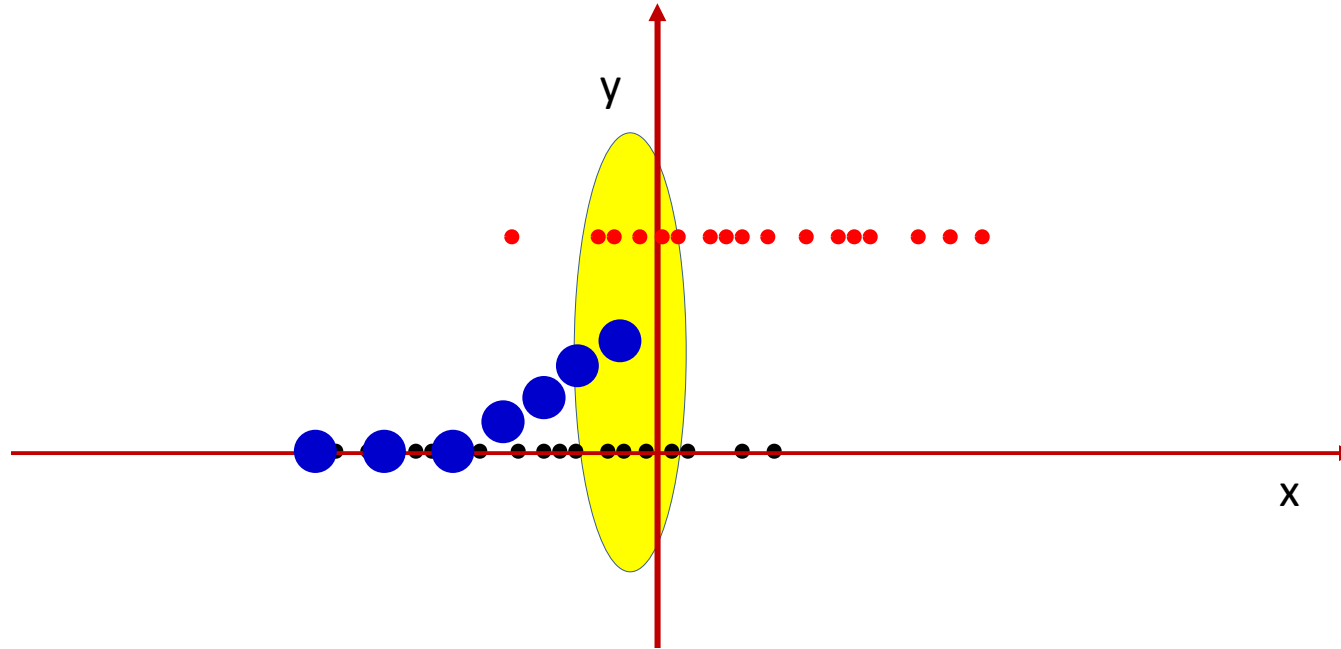
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



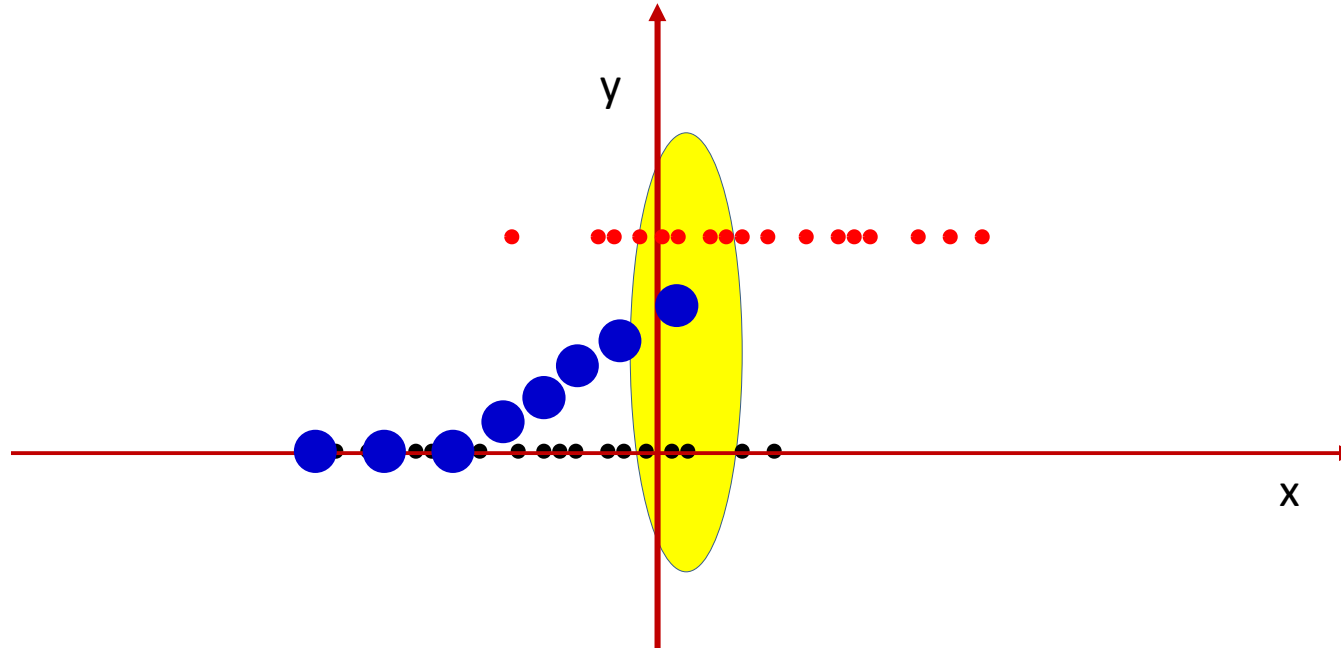
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



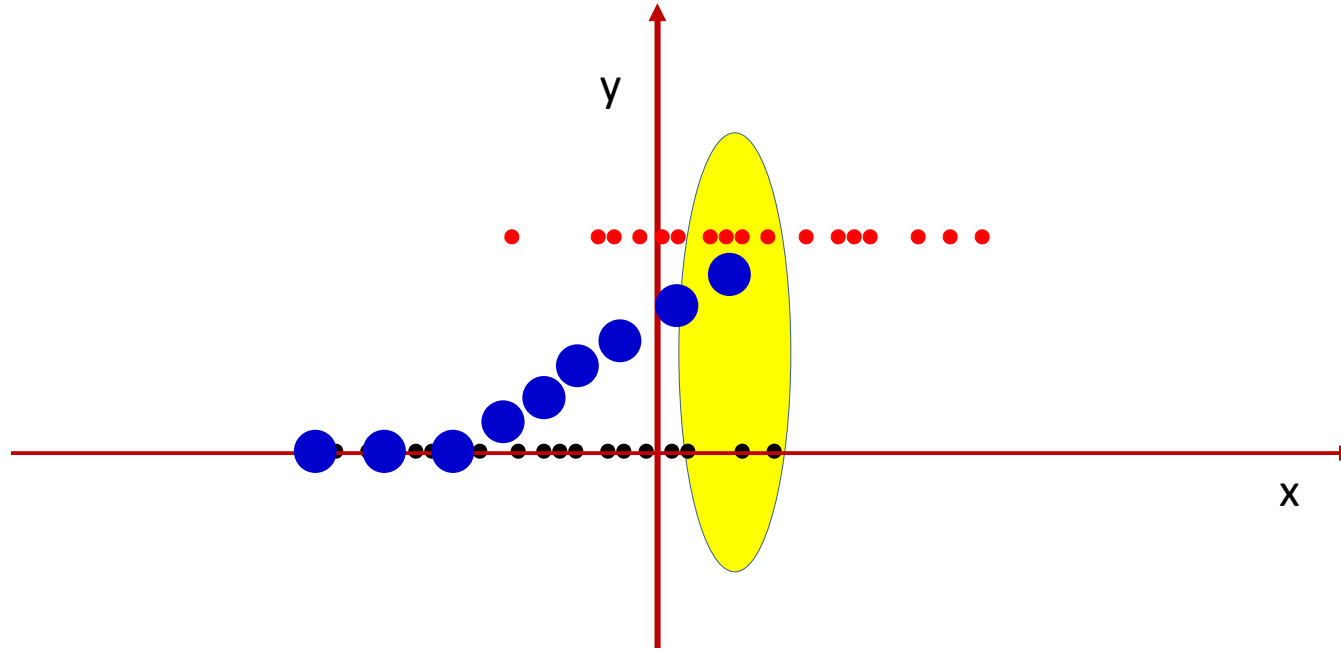
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



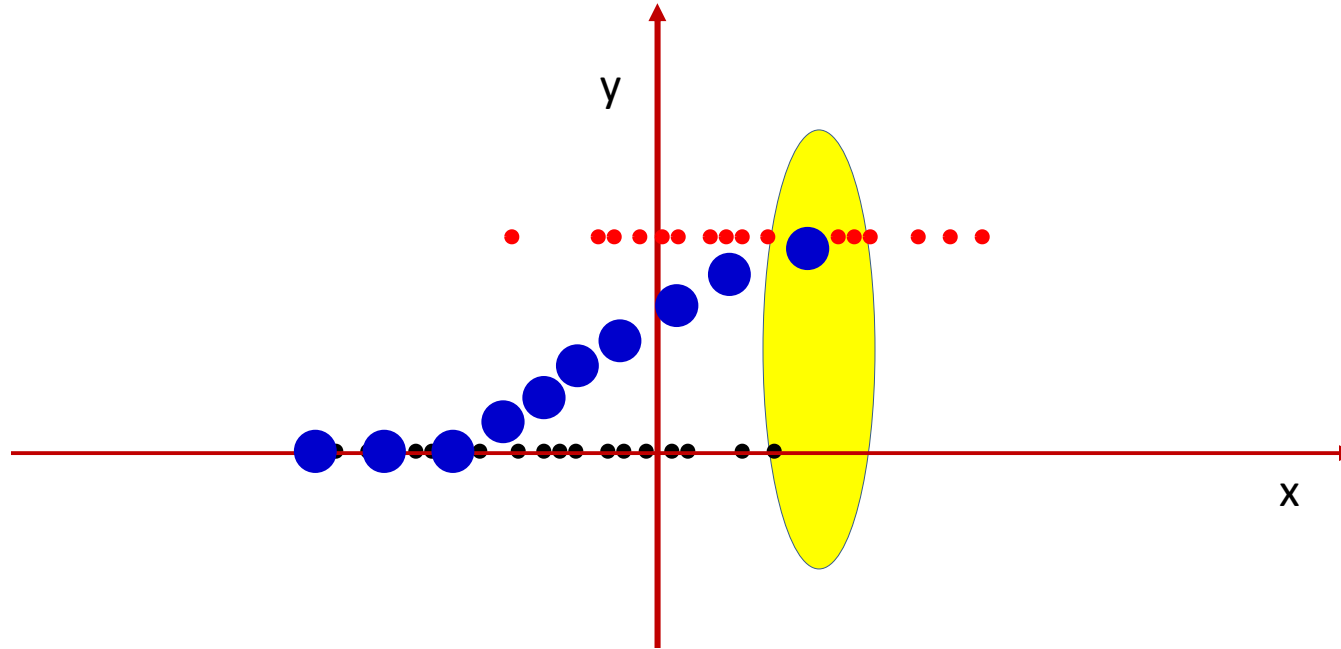
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



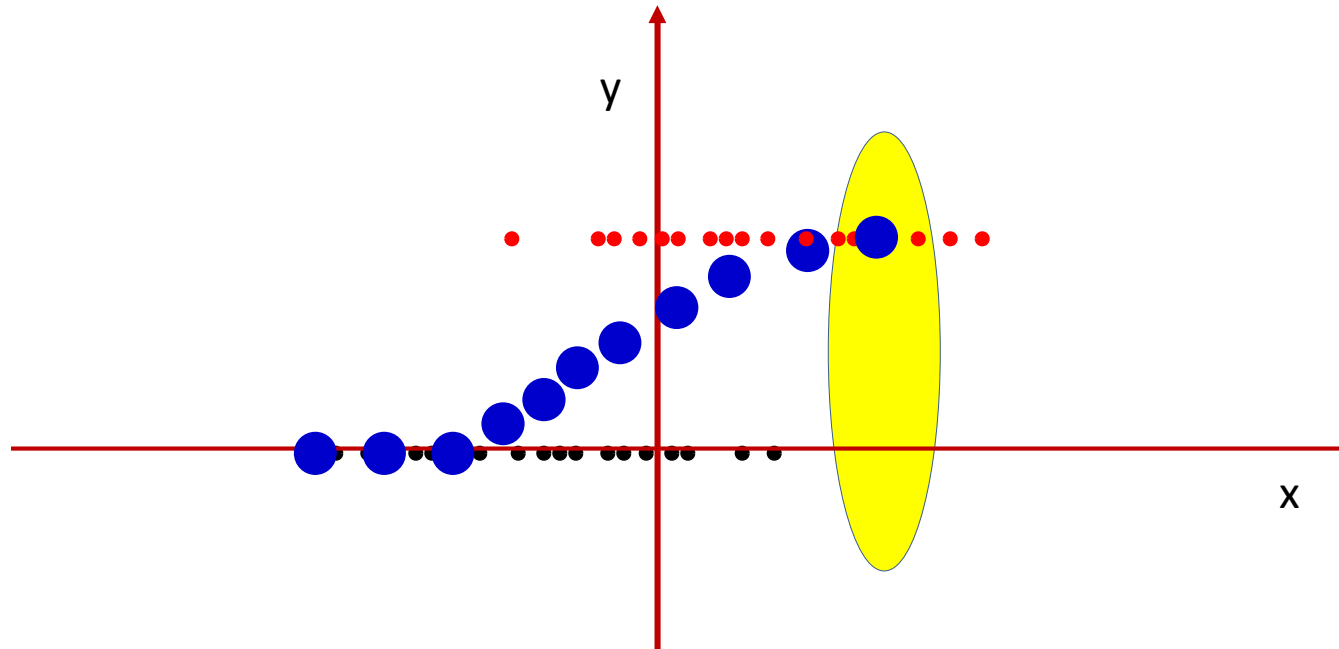
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



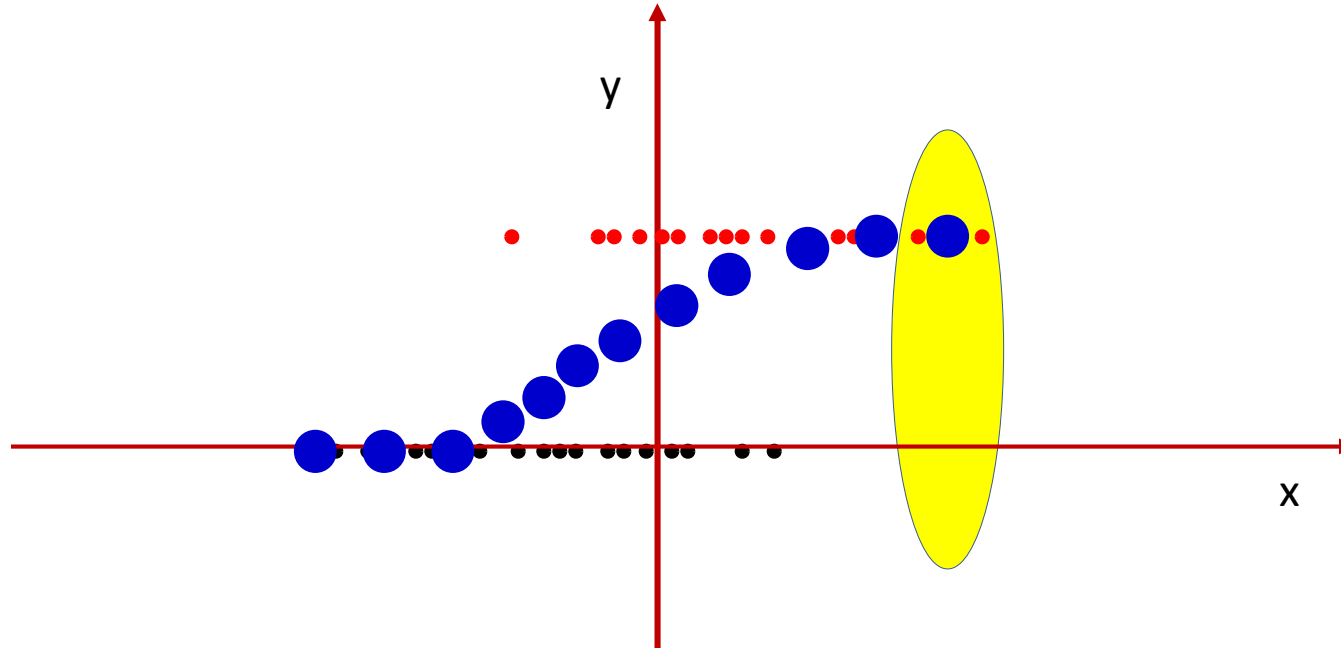
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



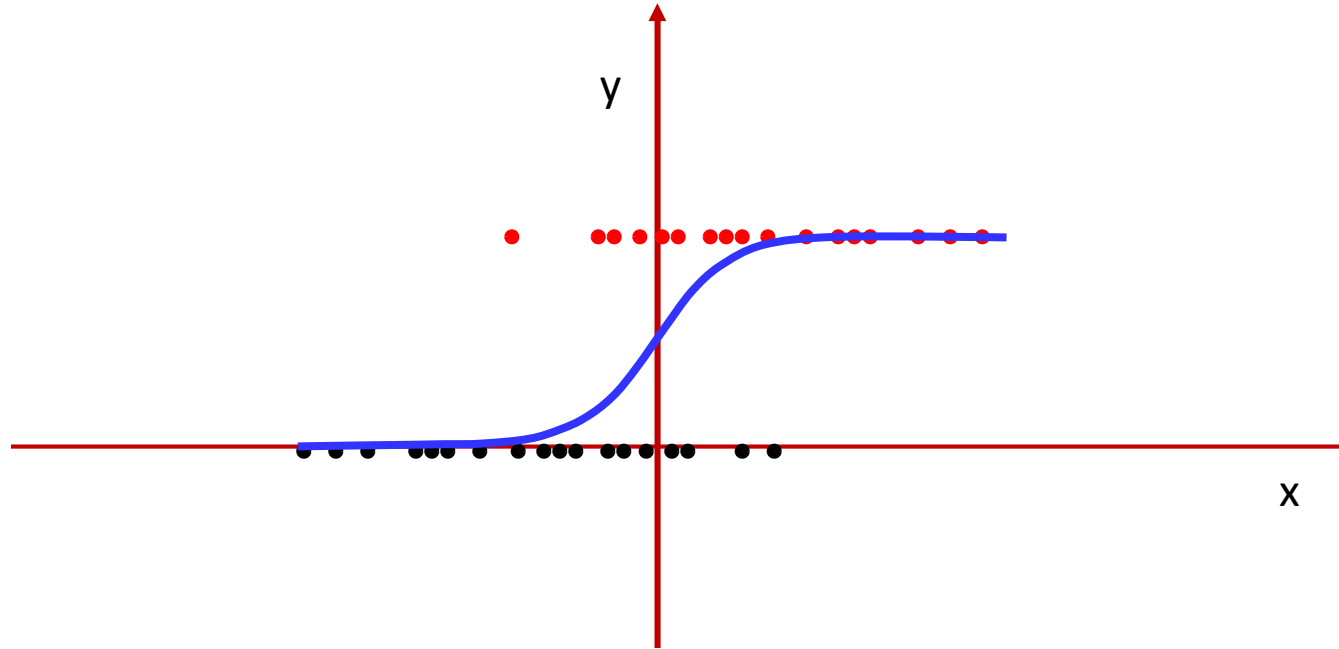
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



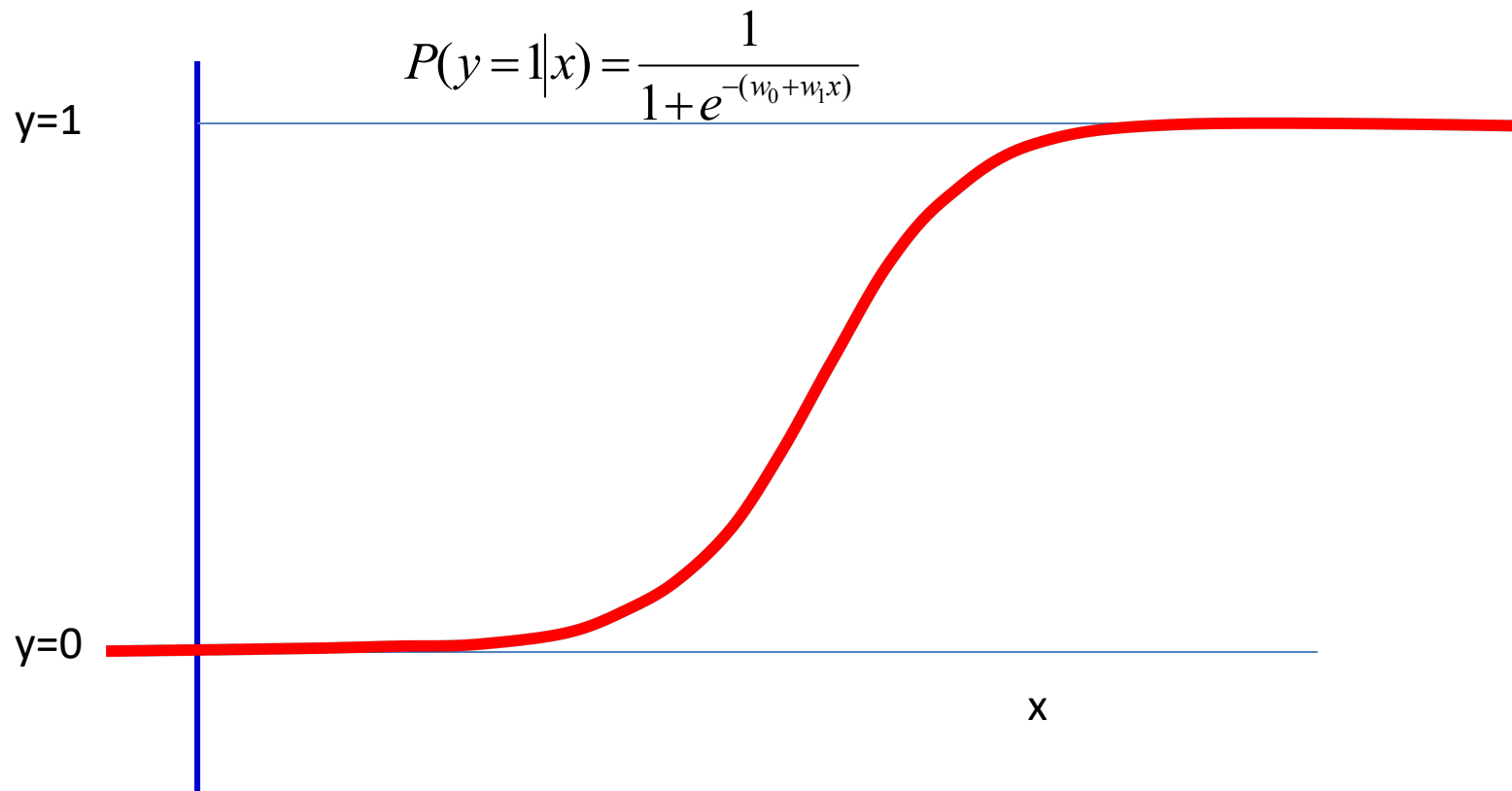
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The *probability* of $y=1$



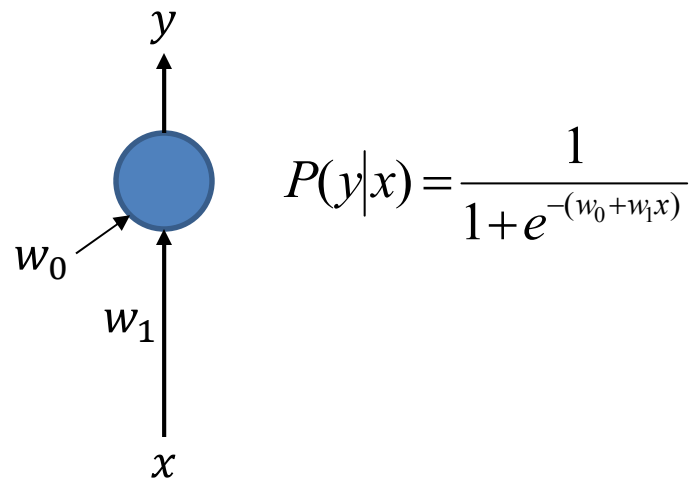
- Consider this differently: at each point look at a small window around that point
- Plot the average value within the window
 - This is an approximation of the *probability* of 1 at that point

The logistic regression model



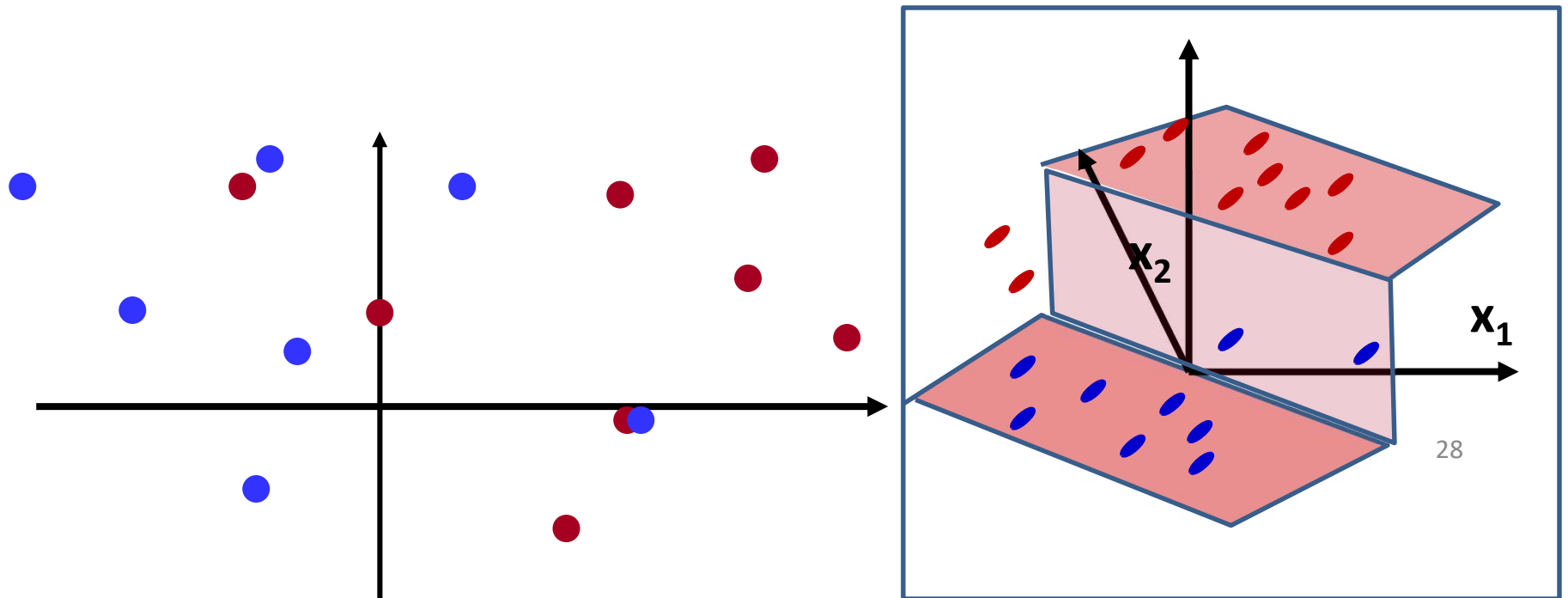
- Class 1 becomes increasingly probable going left to right
 - Very typical in many problems

The logistic perceptron



- A sigmoid perceptron with a single input models the *a posteriori* probability of the class given the input

Linearly inseparable data

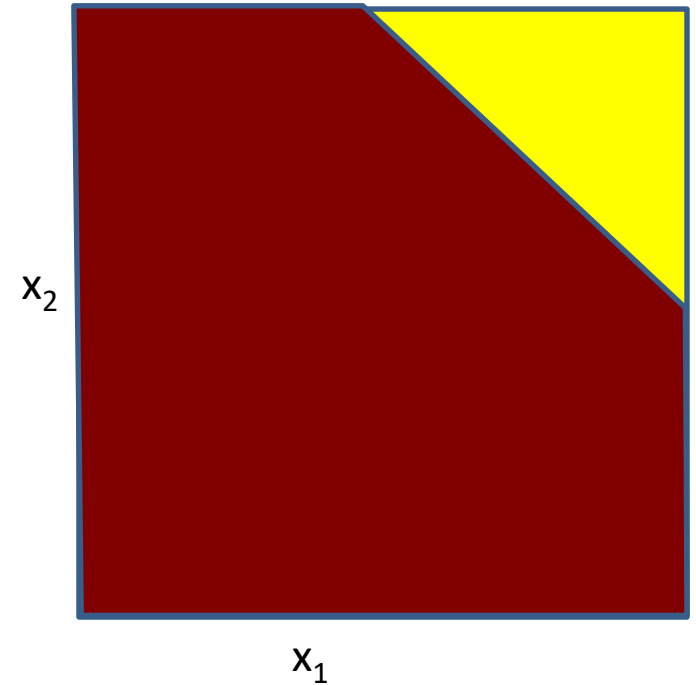
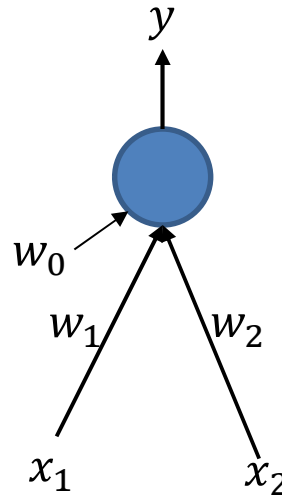
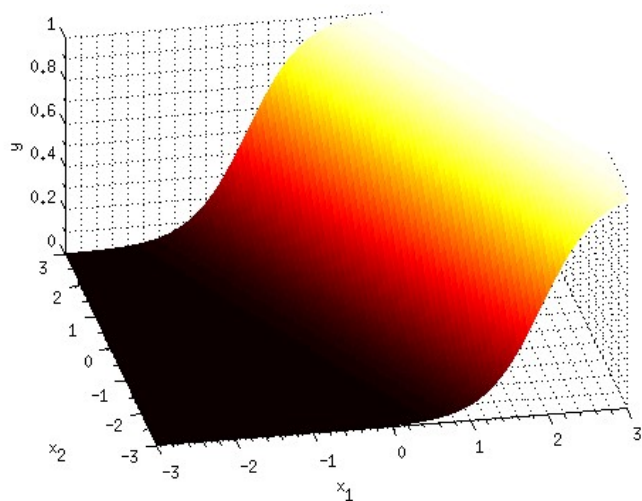


- Two-dimensional example
 - Blue dots (on the floor) on the “red” side
 - Red dots (suspended at $Y=1$) on the “blue” side
 - No line will cleanly separate the two colors

Logistic regression

$$P(Y = 1|X) = \frac{1}{1 + \exp(-(\sum_i w_i x_i + w_0))}$$

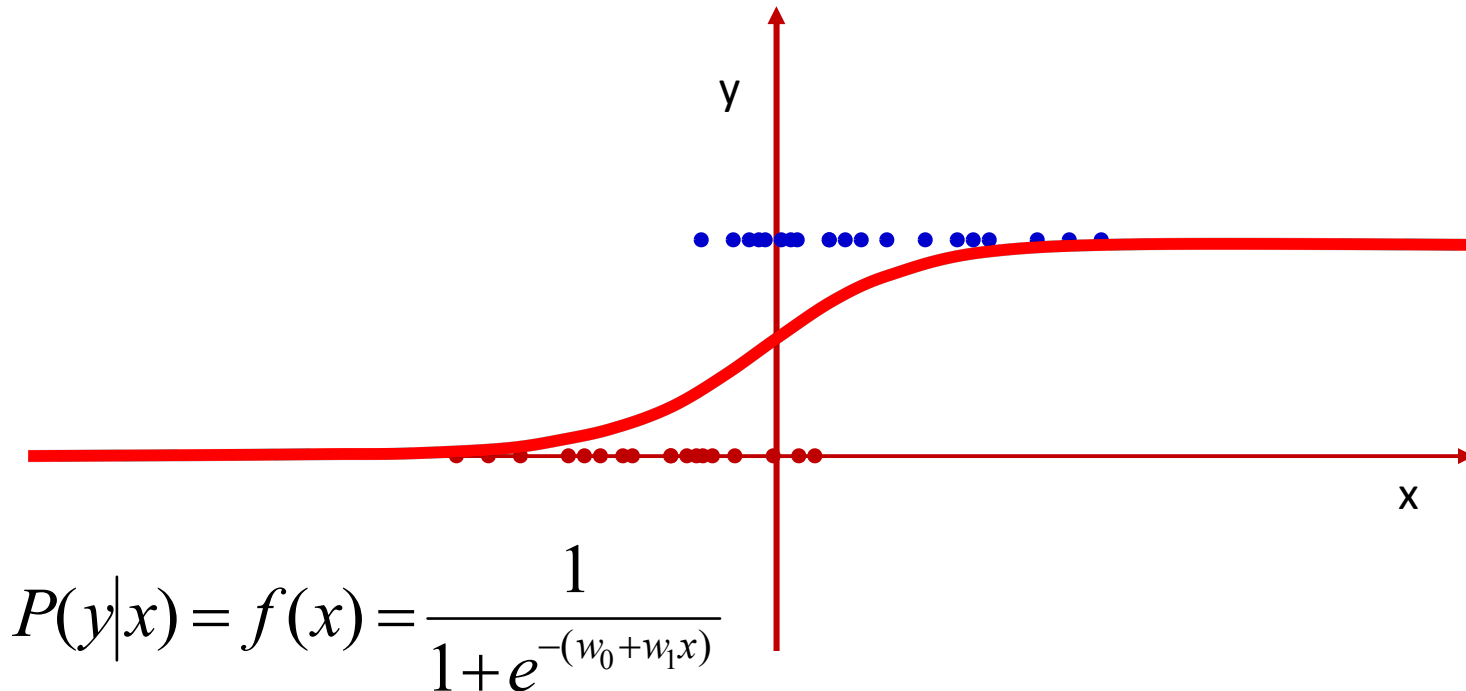
Decision: $y > 0.5$?



When X is a 2-D variable

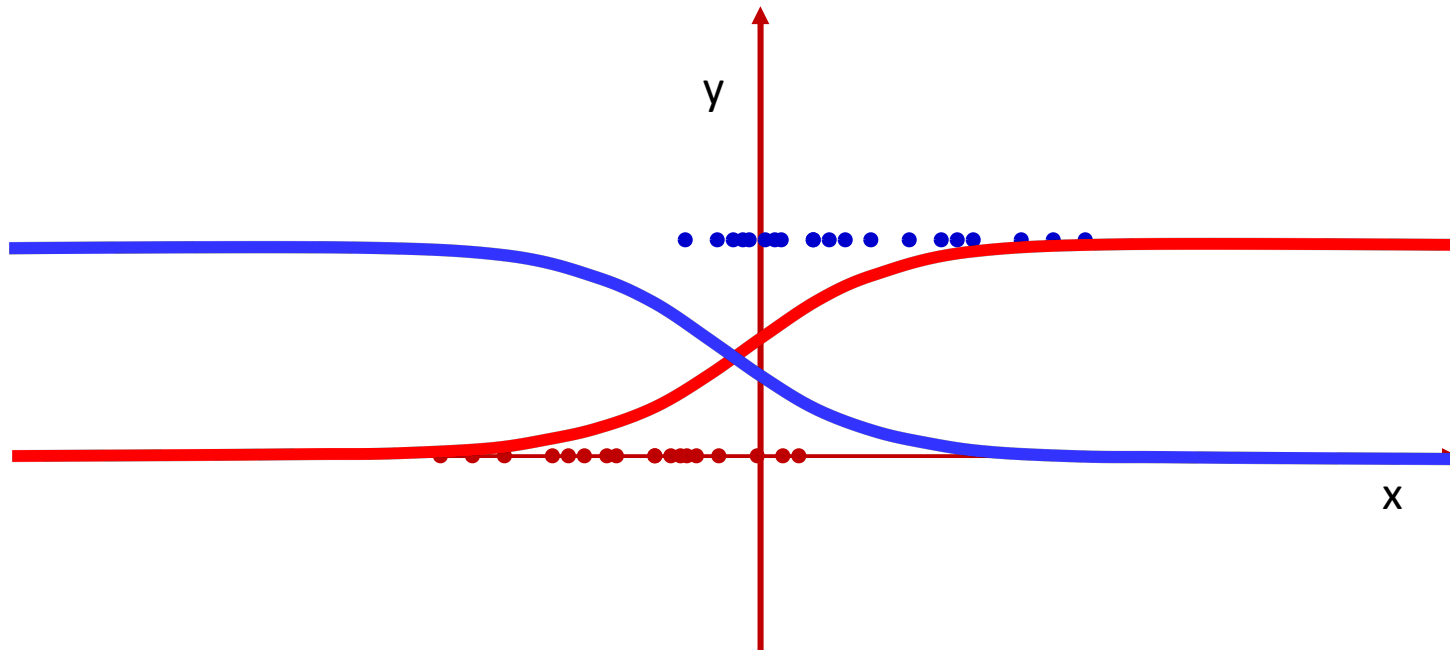
- This is the perceptron with a sigmoid activation
 - It actually computes the *probability* that the input belongs to class 1
 - Decision boundaries may be obtained by comparing the probability to a threshold
 - These boundaries will be lines (hyperplanes in higher dimensions)
 - The sigmoid perceptron is a *linear classifier*

Estimating the model



- Given the training data (many (x, y) pairs represented by the dots), estimate w_0 and w_1 for the curve

Estimating the model



- Easier to represent using a $y = +1/-1$ notation

$$P(y = 1|x) = \frac{1}{1 + e^{-(w_0 + w_1 x)}}$$

$$P(y = -1|x) = \frac{1}{1 + e^{(w_0 + w_1 x)}}$$

$$P(y|x) = \frac{1}{1 + e^{-y(w_0 + w_1 x)}}$$

Estimating the model

- Given: Training data
 $(X_1, y_1), (X_2, y_2), \dots, (X_N, y_N)$
- X s are vectors, y s are binary (0/1) class values
- Total probability of data

$$\begin{aligned} P((X_1, y_1), (X_2, y_2), \dots, (X_N, y_N)) &= \prod_i P(X_i, y_i) \\ &= \prod_i P(X_i) P(y_i | X_i) = \prod_i P(X_i) \frac{1}{1 + e^{-y_i(w_0 + w^T X_i)}} \end{aligned}$$

Estimating the model

- Given: Training data

$$\begin{aligned} P(\text{Training data}) &= \prod_i P(X_i) \frac{1}{1 + e^{-y_i(w_0 + w^T X_i)}} \\ &= \prod_i P(X_i) \prod_i \frac{1}{1 + e^{-y_i(w_0 + w^T X_i)}} \end{aligned}$$

- X s are vectors, y s are binary (0/1) class values
- $\log P(\text{Training data}) =$

$$\sum_i \log P(X_i) + \sum_i \log \left(\frac{1}{1 + e^{-y_i(w_0 + w^T X_i)}} \right)$$

- Total probability of data

Estimating the model

- Log Likelihood

$$\log P(\textit{Training data}) =$$

$$\sum_i \log P(X_i) - \sum_i \log \left(1 + e^{-y_i(w_0 + w^T X_i)} \right)$$

Estimating the model

- Log Likelihood

$$\log P(\textit{Training data}) =$$

$$\sum_i \log P(X_i) - \sum_i \log \left(1 + e^{-y_i(w_0 + w^T X_i)} \right)$$

Maximum likelihood estimation

- Log Likelihood

$$\log P(\textit{Training data}) =$$

$$\sum_i \log P(X_i) - \sum_i \log \left(1 + e^{-y_i(w_0 + w^T X_i)} \right)$$

- Maximum likelihood estimation

$$\hat{w}_0, \hat{w}_1 = \underset{w_0, w_1}{\operatorname{argmax}} \log P(\textit{Training data})$$

- Focusing on the bits that invoke the parameters

$$\hat{w}_0, \hat{w}_1 = \underset{w_0, w_1}{\operatorname{argmax}} \left(- \sum_i \log \left(1 + e^{-y_i(w_0 + w^T X_i)} \right) \right)$$

Maximum Likelihood Estimate

- Equals (note argmin rather than argmax)

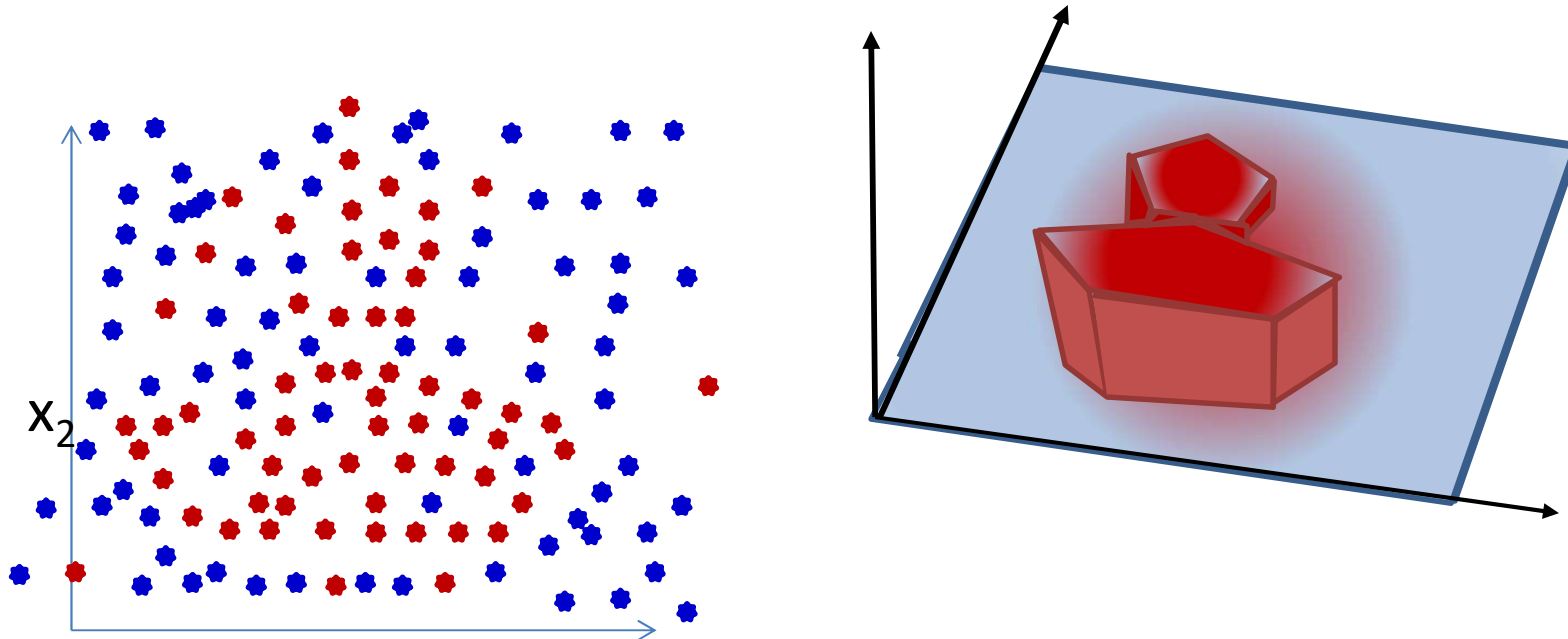
$$\hat{w}_0, \hat{w}_1 = \underset{w_0, w}{\operatorname{argmin}} \sum_i \log \left(1 + e^{-y_i(w_0 + w^T X_i)} \right)$$

- Identical to minimizing the KL divergence between the desired output y and actual output

$$\frac{1}{1 + e^{-(w_0 + w^T X_i)}}$$

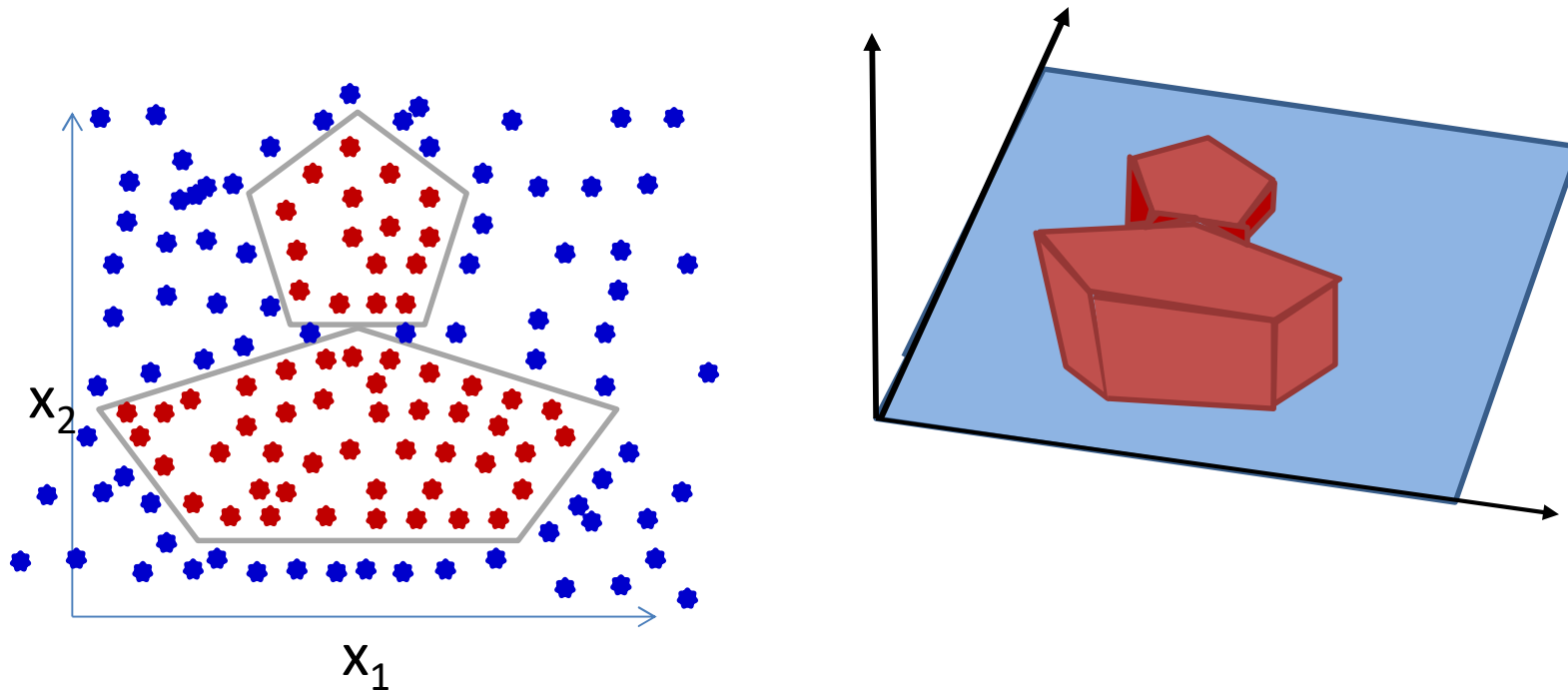
- Cannot be solved directly, needs gradient descent

So what about this one?



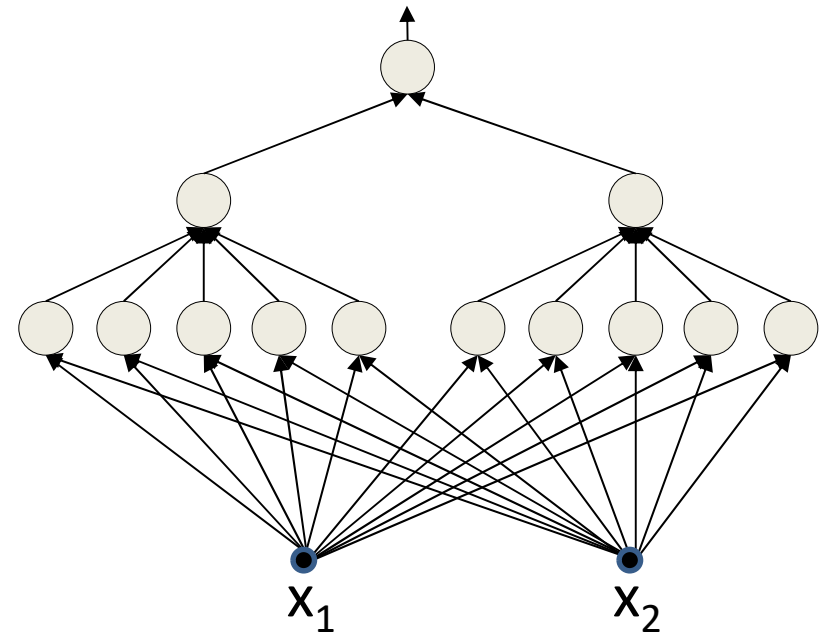
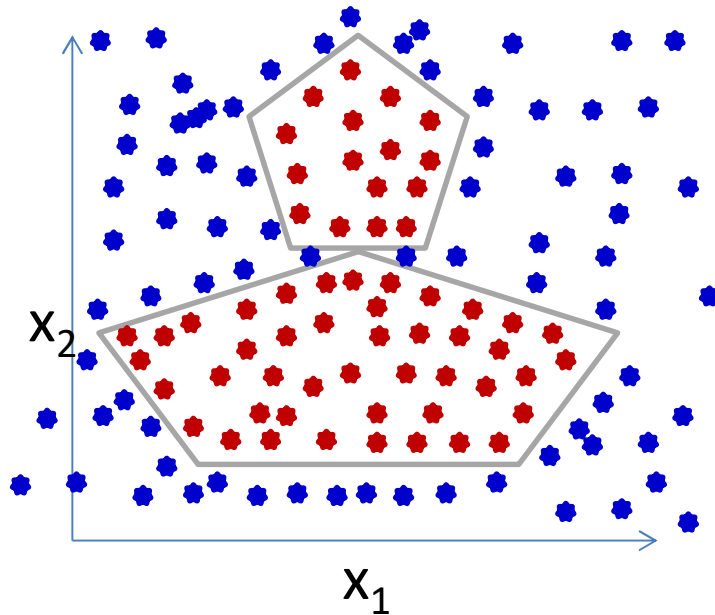
- Non-linear classifiers..

First consider the separable case..



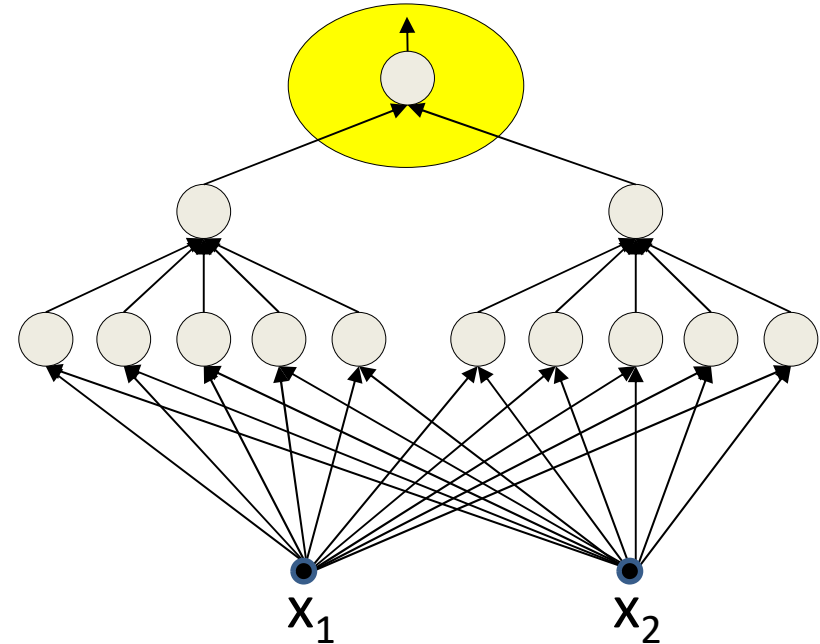
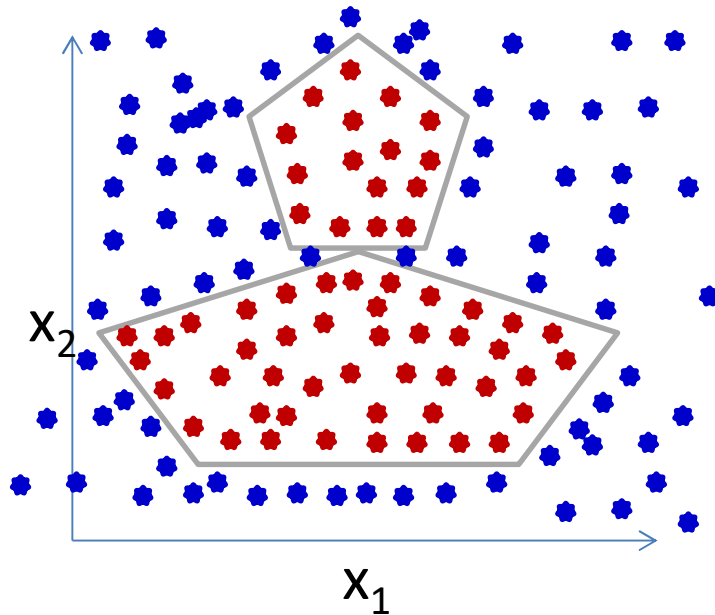
- When the net must learn to classify..

First consider the separable case..



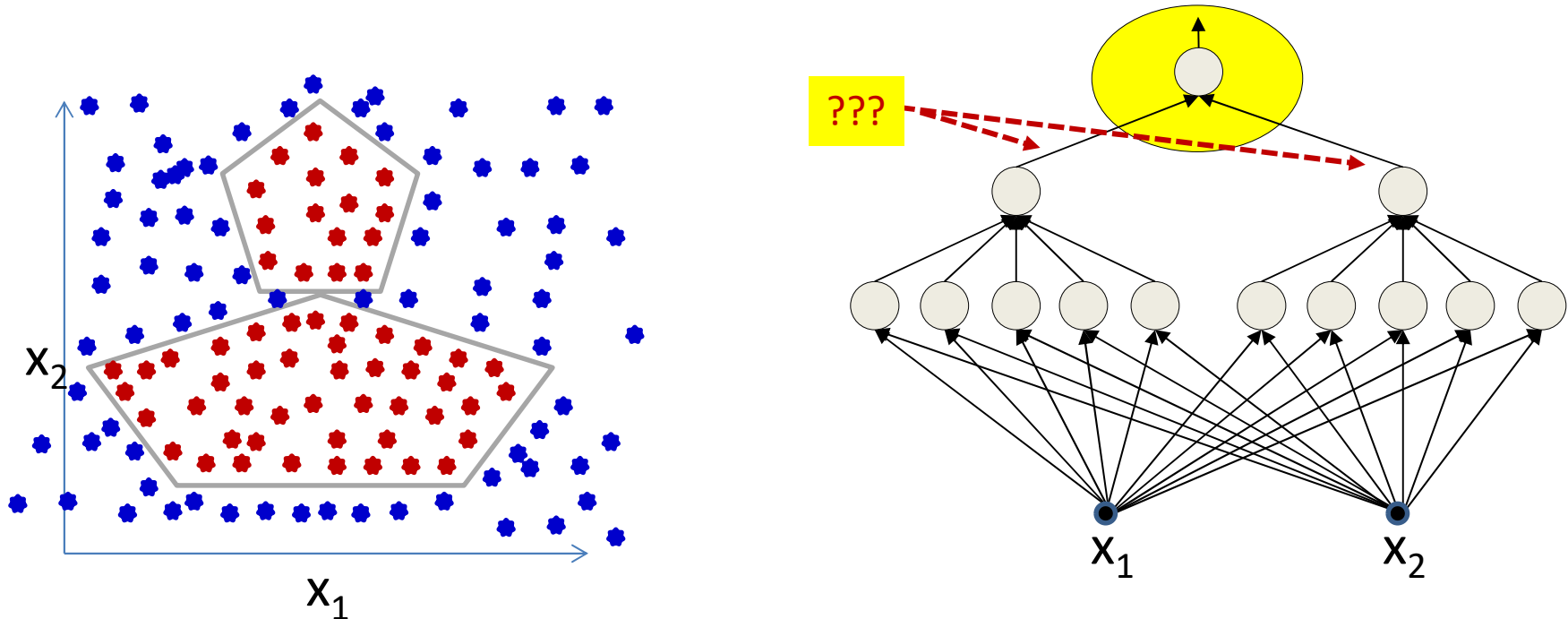
- For a “sufficient” net

First consider the separable case..



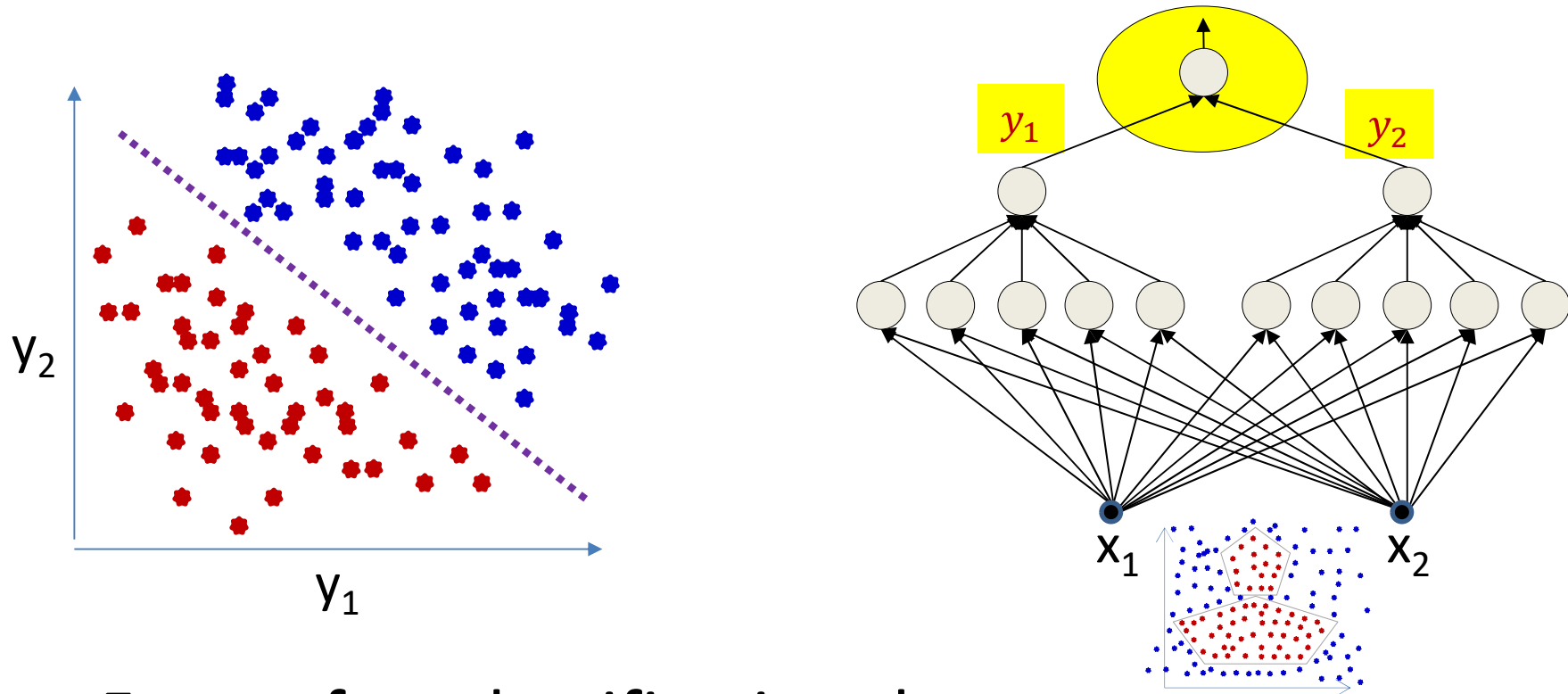
- For a “sufficient” net
- This final perceptron is a linear classifier

First consider the separable case..



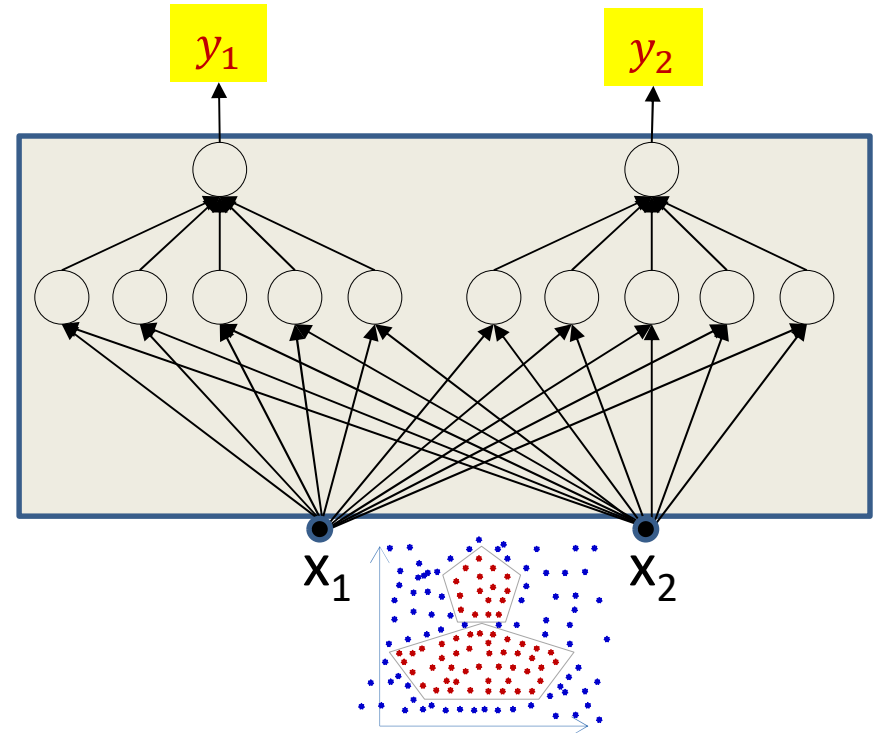
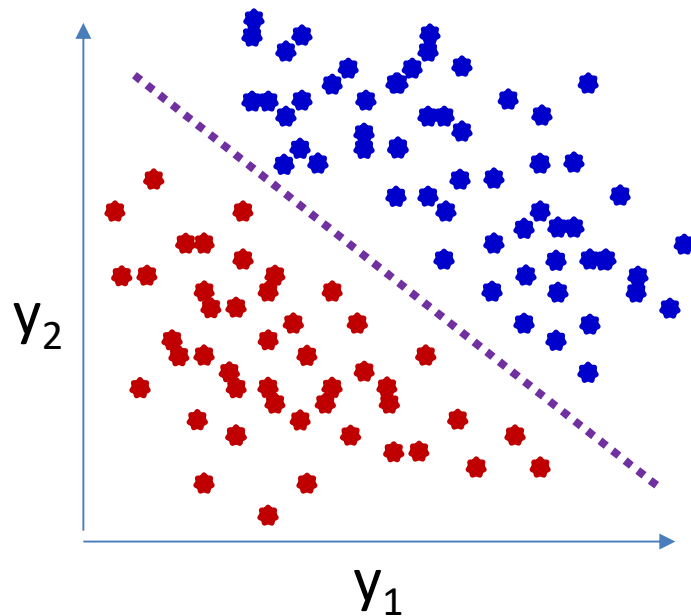
- For a “sufficient” net
- This final perceptron is a linear classifier over the output of the penultimate layer

First consider the separable case..



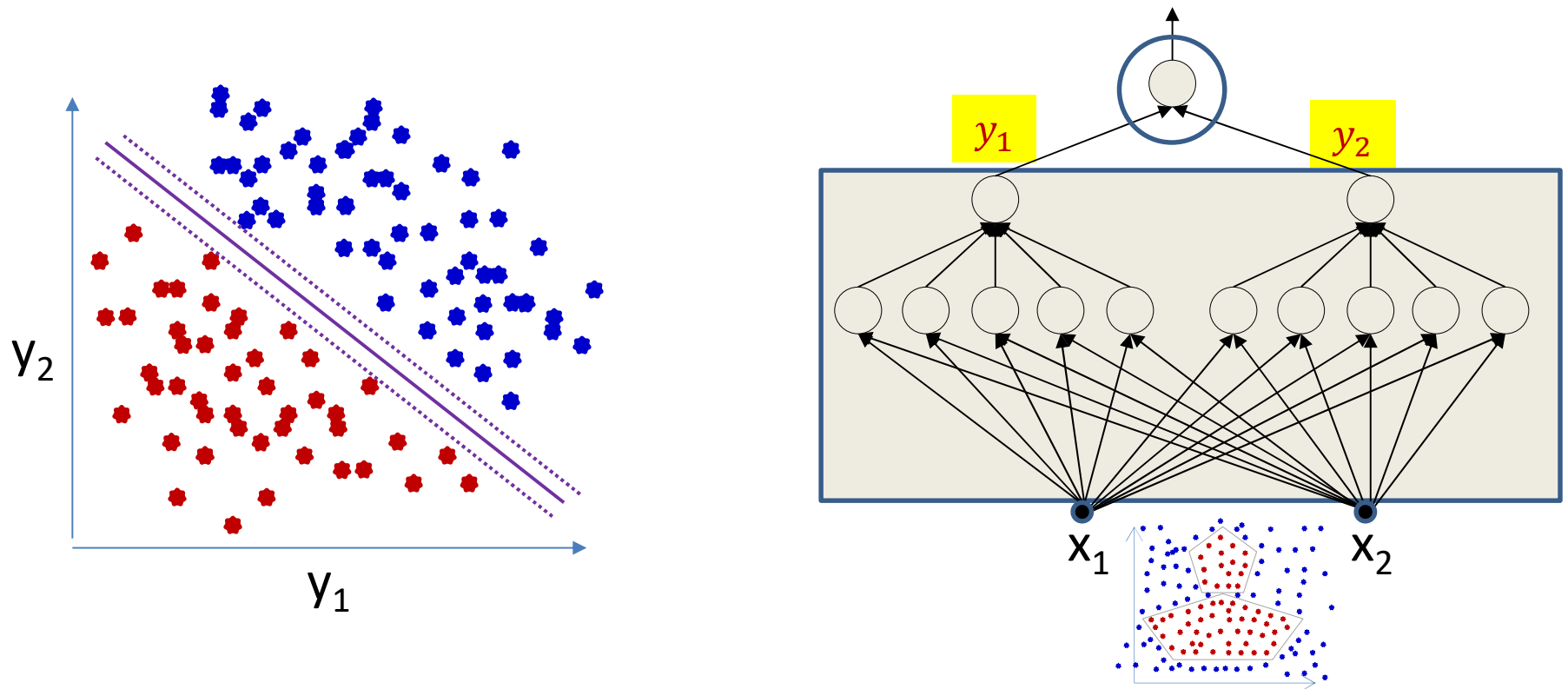
- For perfect classification the output of the penultimate layer must be linearly separable

First consider the separable case..



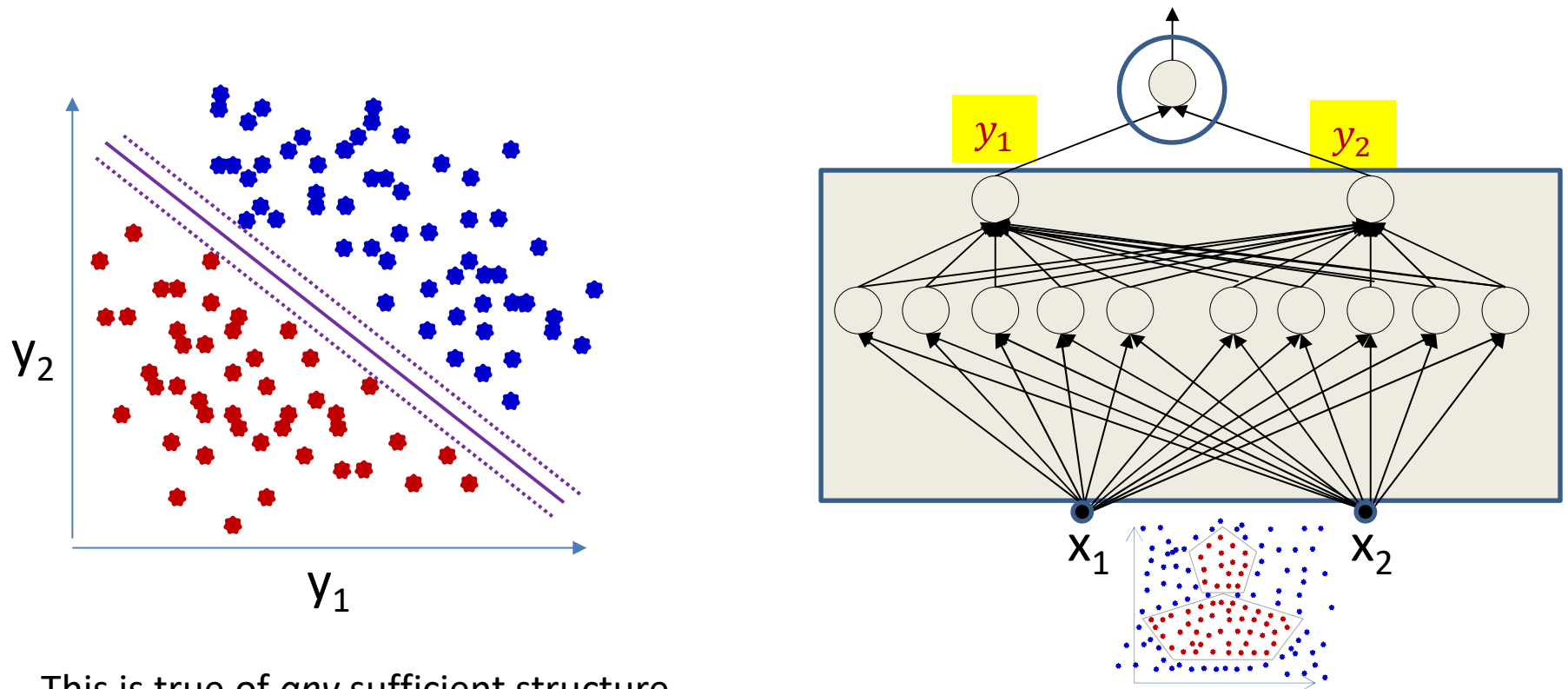
- The rest of the network may be viewed as a transformation that transforms data from non-linear classes to linearly separable features

First consider the separable case..



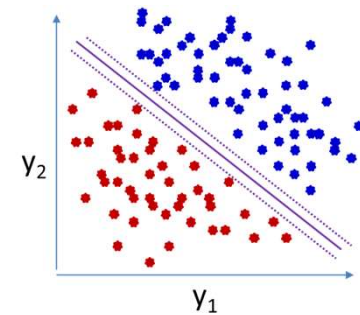
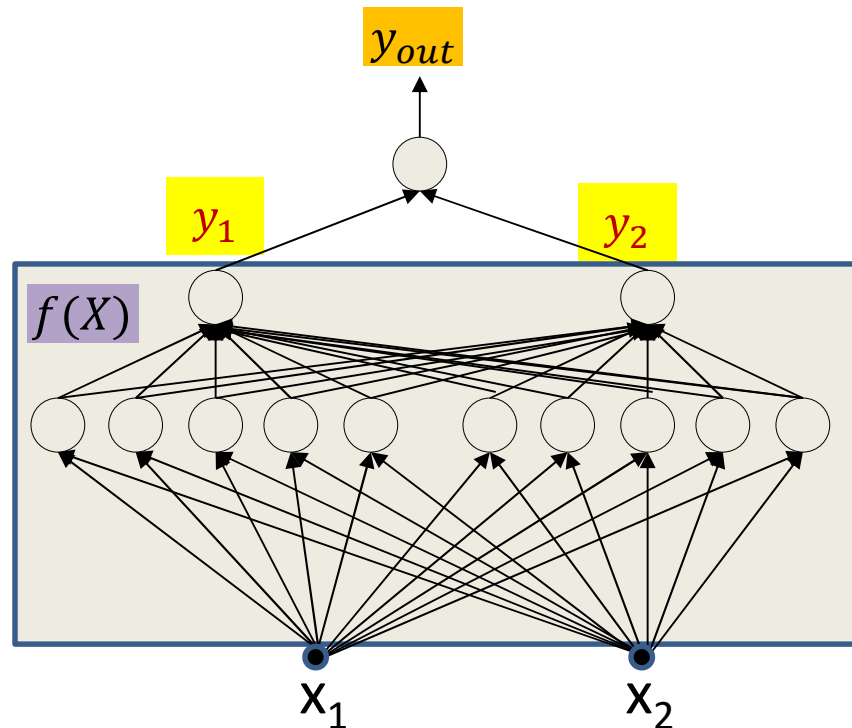
- The rest of the network may be viewed as a transformation that transforms data from non-linear classes to linearly separable features
 - We can now attach *any* linear classifier above it for perfect classification
 - Need not be a perceptron
 - In fact, for **binary** classifiers an SVM on top of the features may be more generalizable!

First consider the separable case..



- This is true of *any* sufficient structure
 - Not just the optimal one
- For *insufficient* structures, the network may *attempt* to transform the inputs to linearly separable features
 - Will fail to separate
 - Still, for binary problems, using an SVM with slack may be more effective than a final perceptron!

Mathematically..

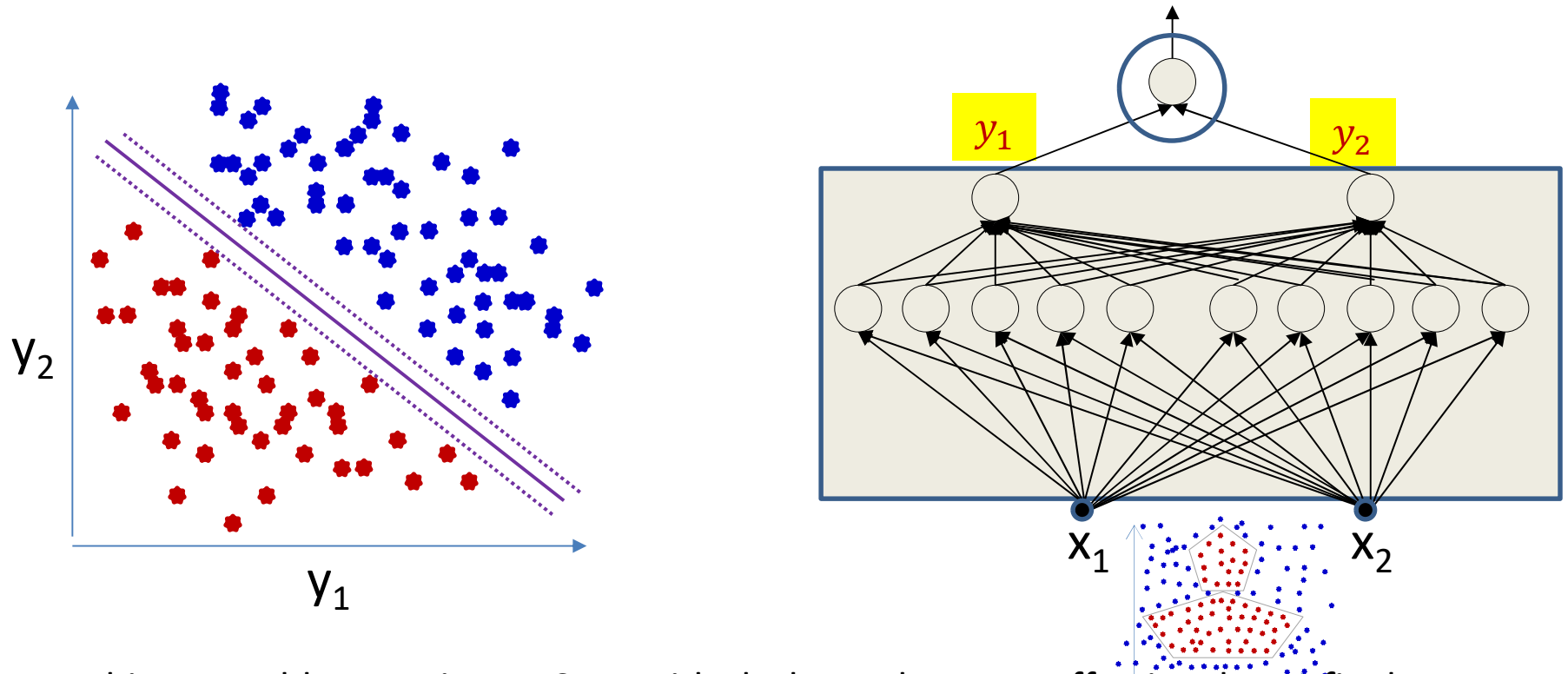


- $$y_{out} = \frac{1}{1+\exp(b+W^T Y)} = \frac{1}{1+\exp(b+W^T f(X))}$$
- The data are (almost) linearly separable in the space of Y
- The network until the second-to-last layer is a non-linear function $f(X)$ that converts the input space of X into the feature space Y where the classes are maximally linearly separable

Story so far

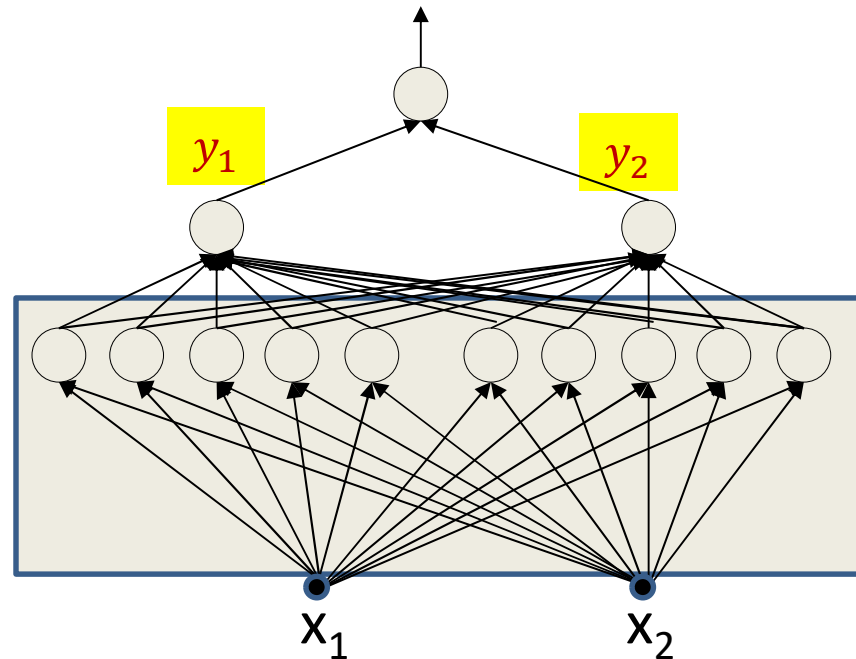
- A classification MLP actually comprises two components
 - A “feature extraction network” that converts the inputs into linearly separable features
 - Or *nearly* linearly separable features
 - A final linear classifier that operates on the linearly separable features

An SVM at the output?



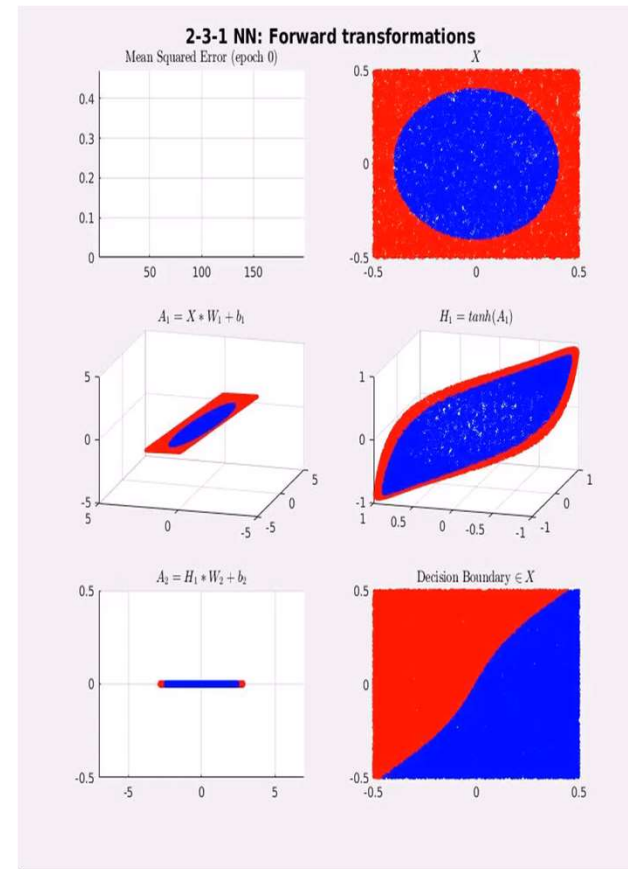
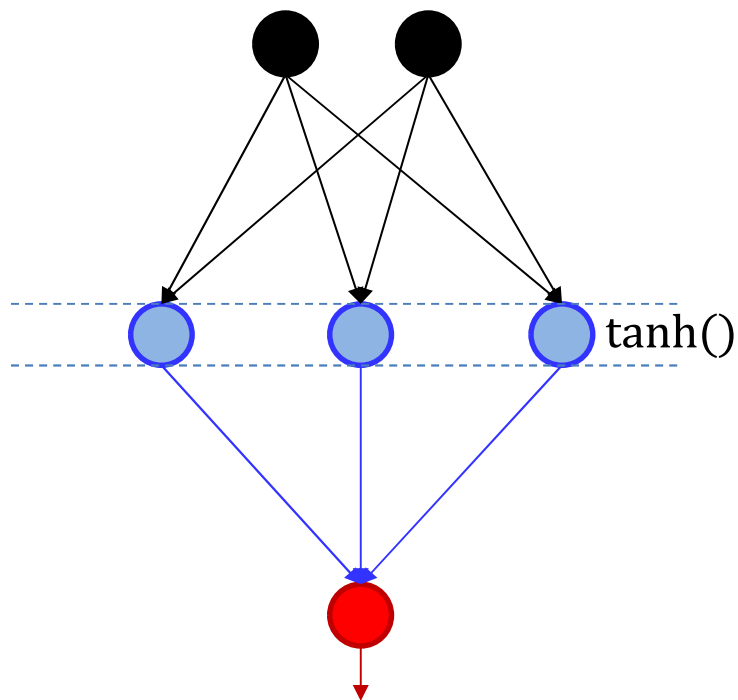
- For binary problems, using an SVM with slack may be more effective than a final perceptron!
- How does that work??
 - *Option 1:* First train the MLP with a perceptron at the output, then detach the feature extraction, compute features, and train an SVM
 - *Option 2:* Directly employ a max-margin rule at the output, and optimize the entire network
 - Left as an exercise for the curious

How about the lower layers?



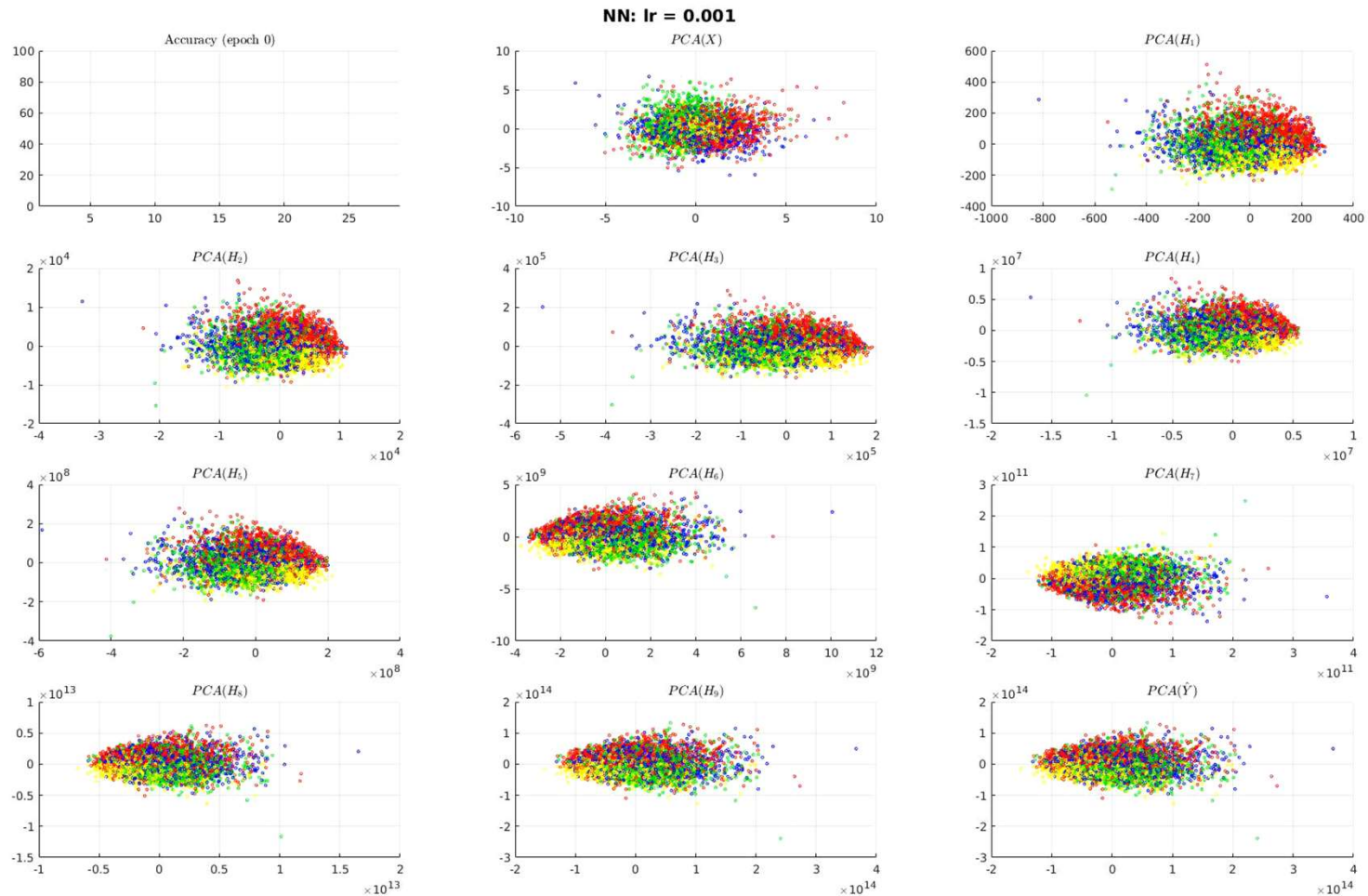
- How do the lower layers respond?
 - They too compute features
 - But how do they look
- Manifold hypothesis: For separable classes, the classes are linearly separable on a non-linear manifold
- Layers sequentially “straighten” the data manifold
 - Until the final layer, which fully linearizes it

The behavior of the layers



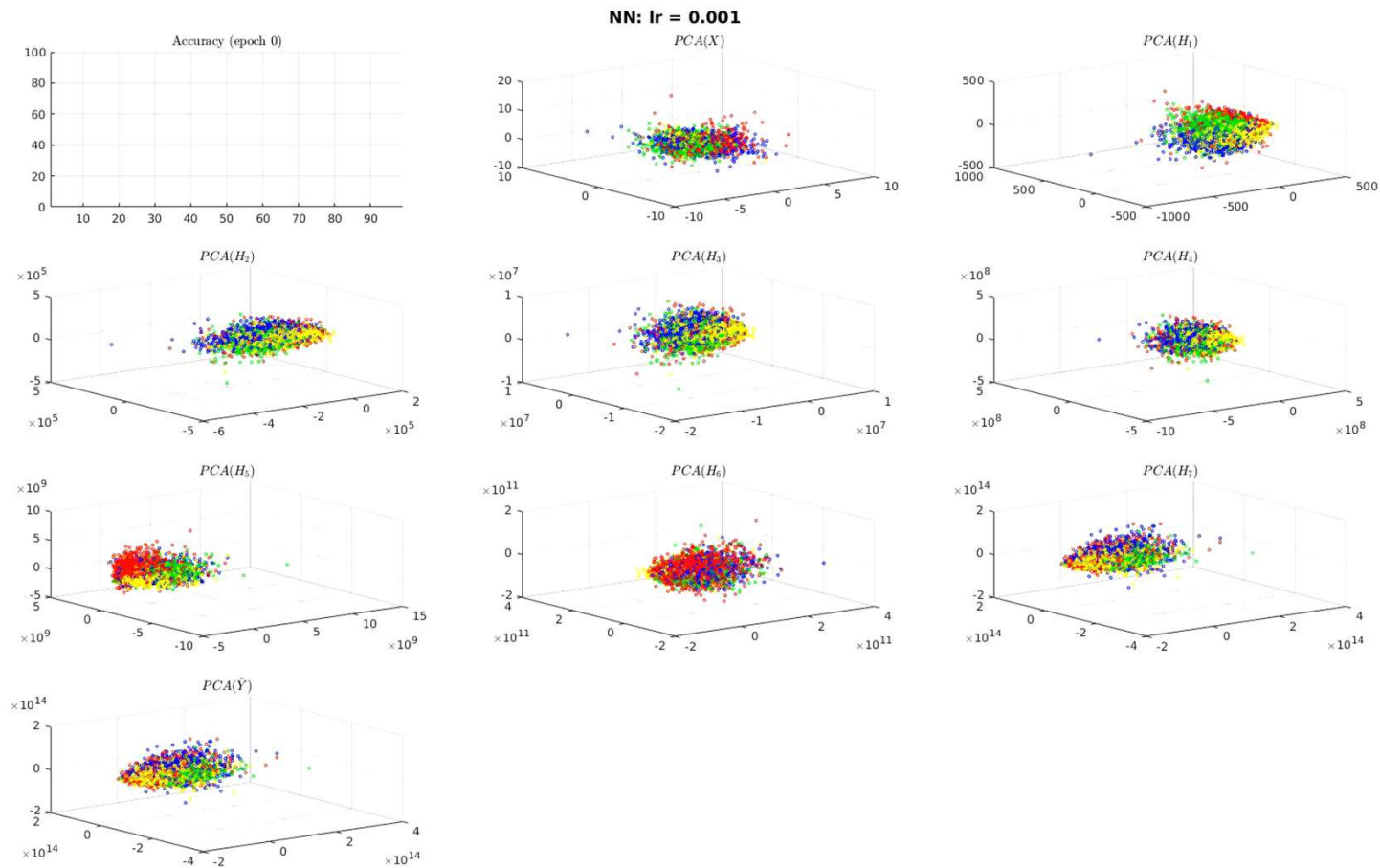
- Synthetic example: Feature space

The behavior of the layers



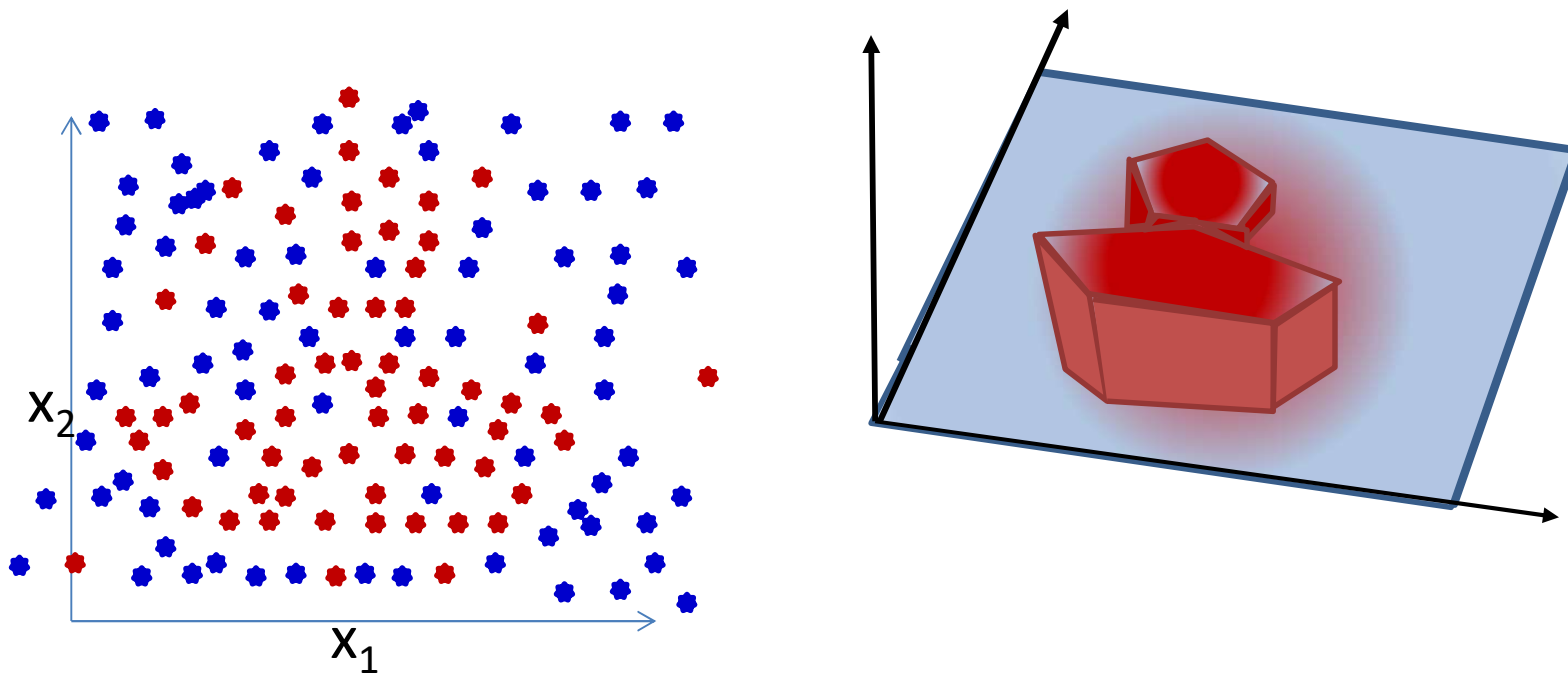
- CIFAR

The behavior of the layers



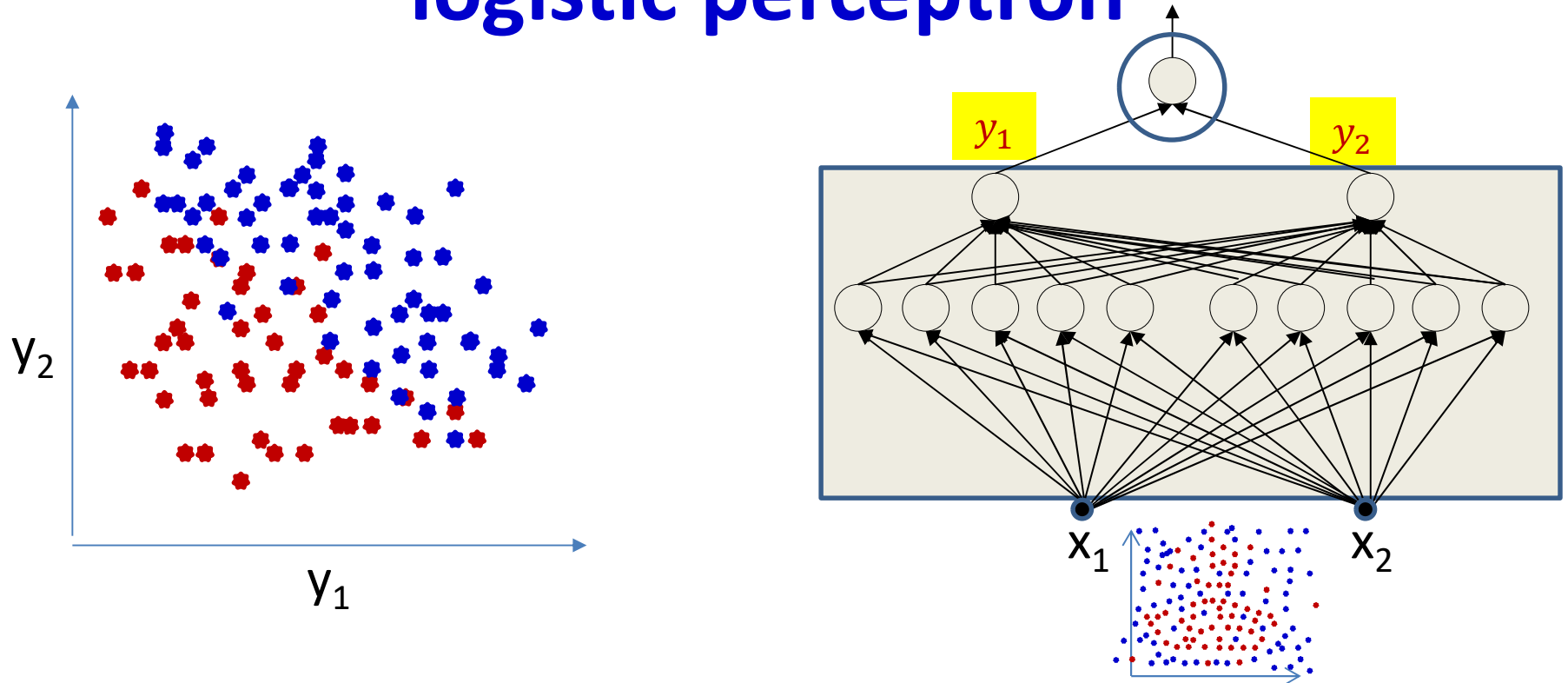
- CIFAR

When the data are not separable and boundaries are not linear..



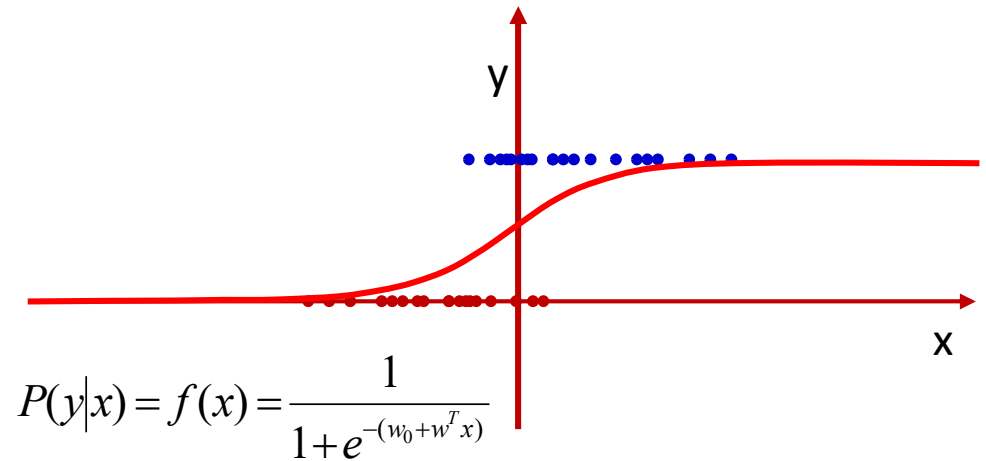
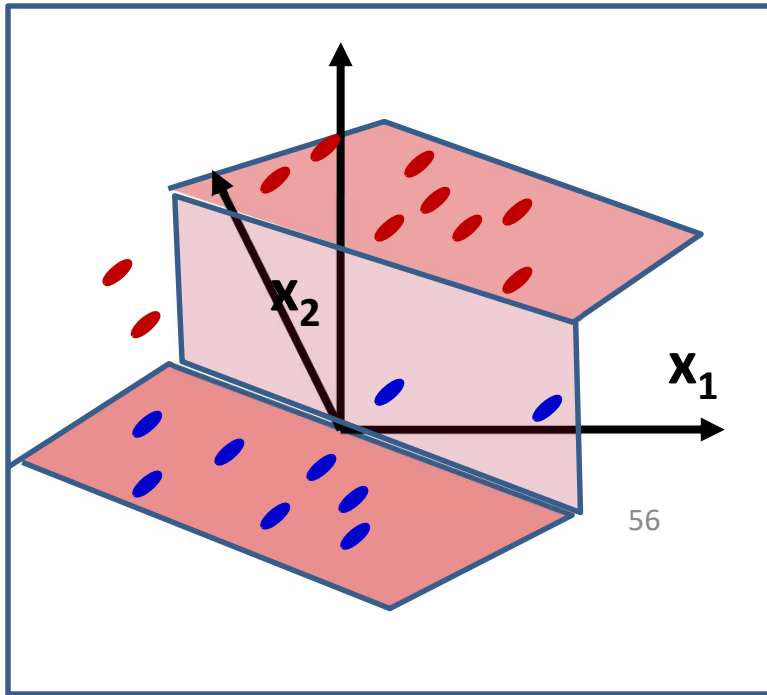
- More typical setting for classification problems

Inseparable classes with an output logistic perceptron



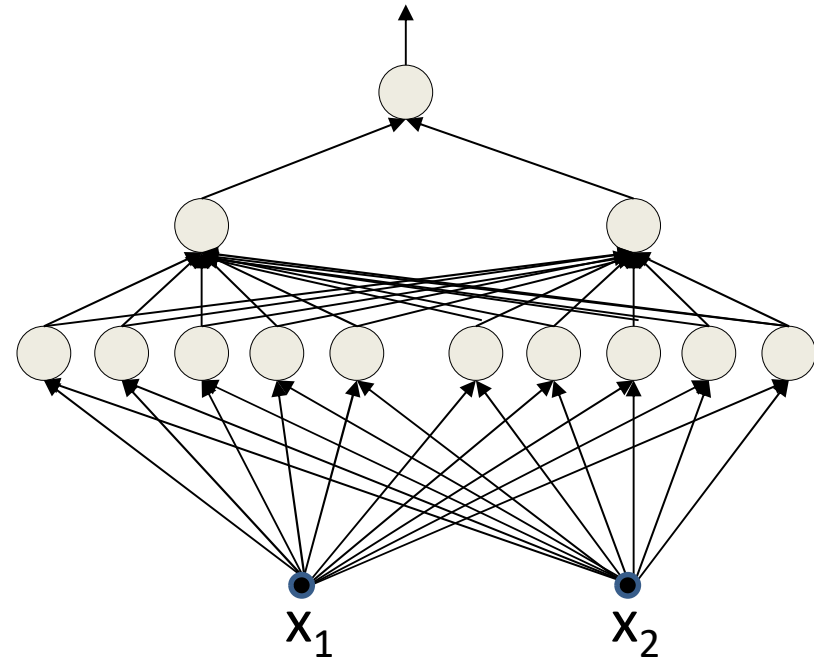
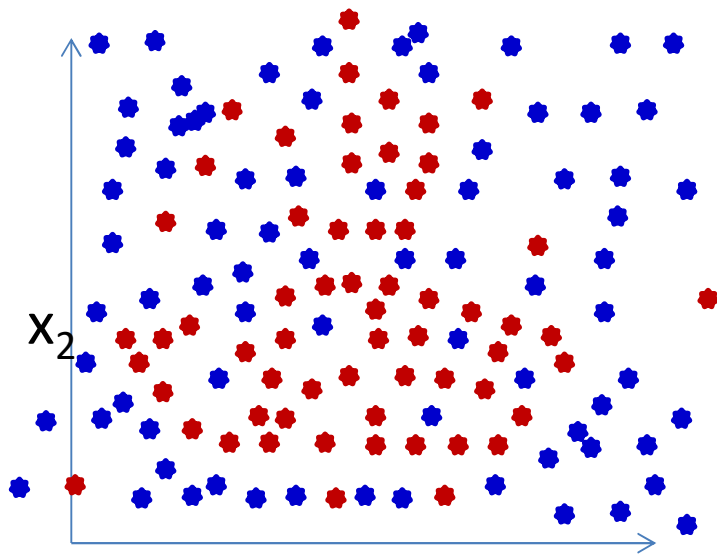
- The “feature extraction” layer transforms the data such that the posterior probability may now be modelled by a logistic

Inseparable classes with an output logistic perceptron



- The “feature extraction” layer transforms the data such that the posterior probability may now be modelled by a logistic
 - The output logistic computes the posterior probability of the class given the input

When the data are not separable and boundaries are not linear..



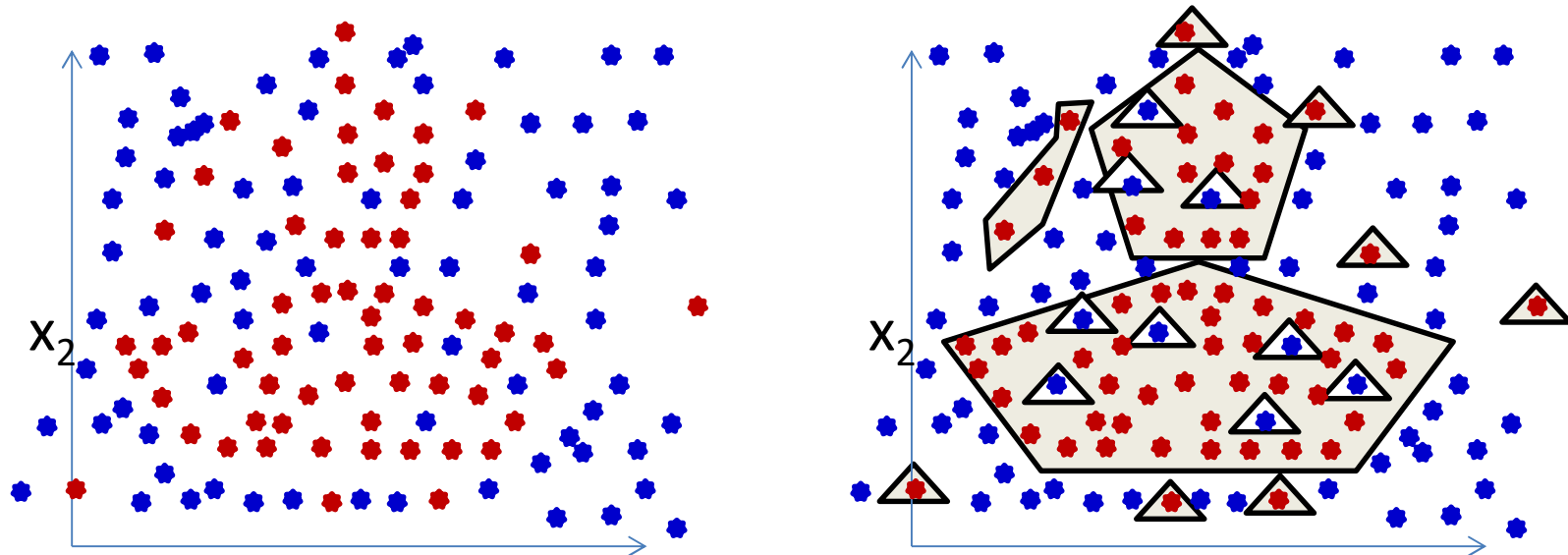
- The output of the network is $P(y|x)$
 - For multi-class networks, it will be the *vector* of a posteriori class probabilities

Everything in this book may be wrong!

– Richard Bach (Illusions)



There's no such thing as inseparable classes



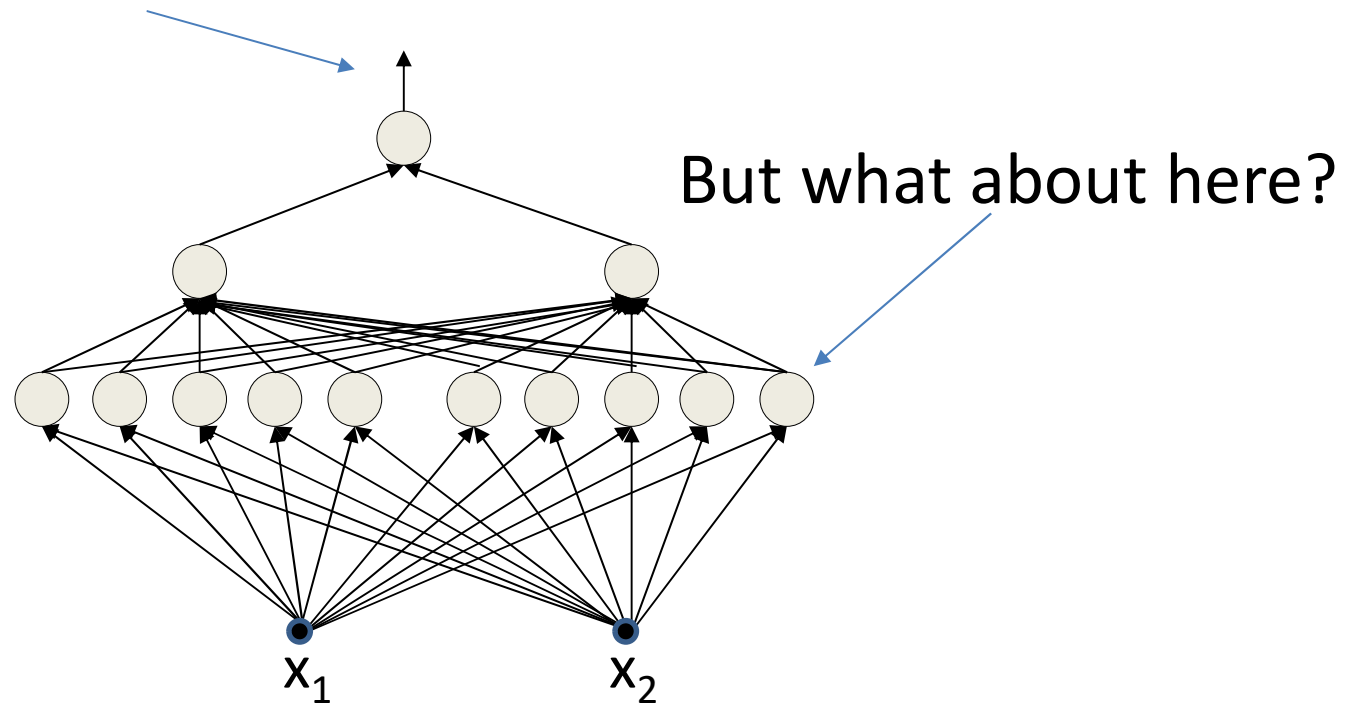
- A sufficiently detailed architecture can separate nearly *any* arrangement of points
 - “Correctness” of the suggested intuitions subject to various parameters, such as regularization, detail of network, training paradigm, convergence etc..

Changing gears..

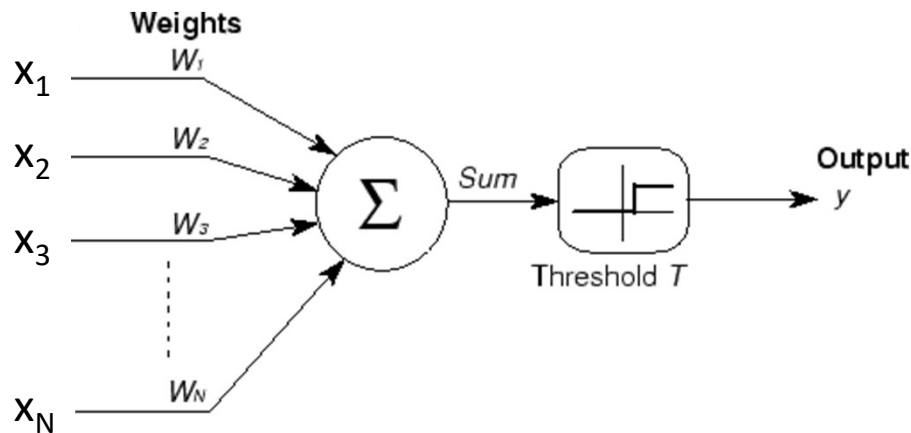


Intermediate layers

We've seen what the network learns here



Recall: The basic perceptron

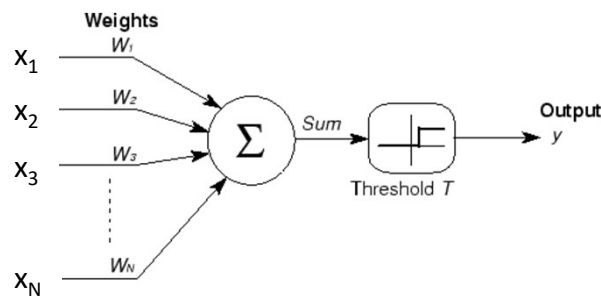


$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i \geq T \\ 0 & \text{else} \end{cases}$$

$$y = \begin{cases} 1 & \text{if } \mathbf{x}^T \mathbf{w} \geq T \\ 0 & \text{else} \end{cases}$$

- What do the *weights* tell us?
 - The neuron fires if the inner product between the weights and the inputs exceeds a threshold

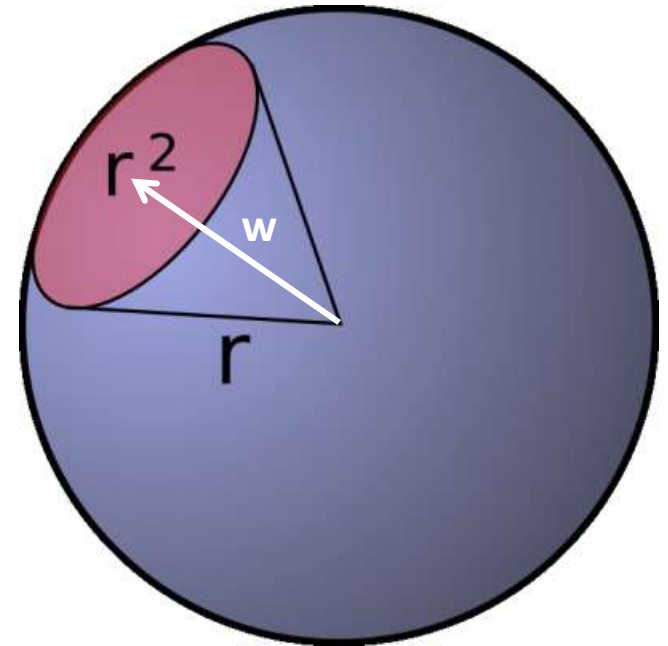
Recall: The weight as a “template”



$$X^T W > T$$

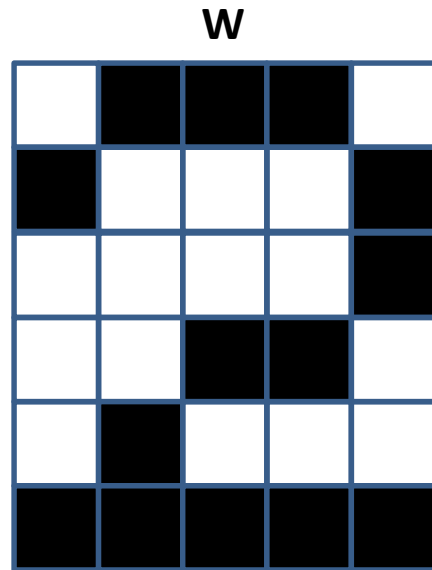
$$\cos \theta > \frac{T}{|W||X|}$$

$$\theta < \cos^{-1} \left(\frac{T}{|W||X|} \right)$$

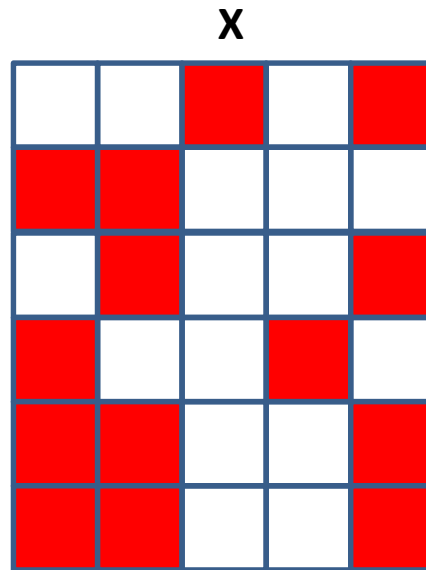


- The perceptron fires if the input is within a specified angle of the weight
 - Represents a convex region on the surface of the sphere!
 - The network is a Boolean function over these regions.
 - The overall decision region can be arbitrarily nonconvex
- Neuron fires if the input vector is close enough to the weight vector.
 - If the input pattern matches the weight pattern closely enough

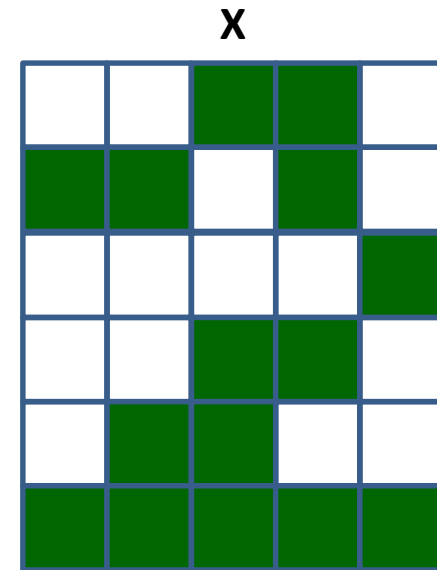
Recall: The weight as a template



$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i \geq T \\ 0 & \text{else} \end{cases}$$



Correlation = 0.57

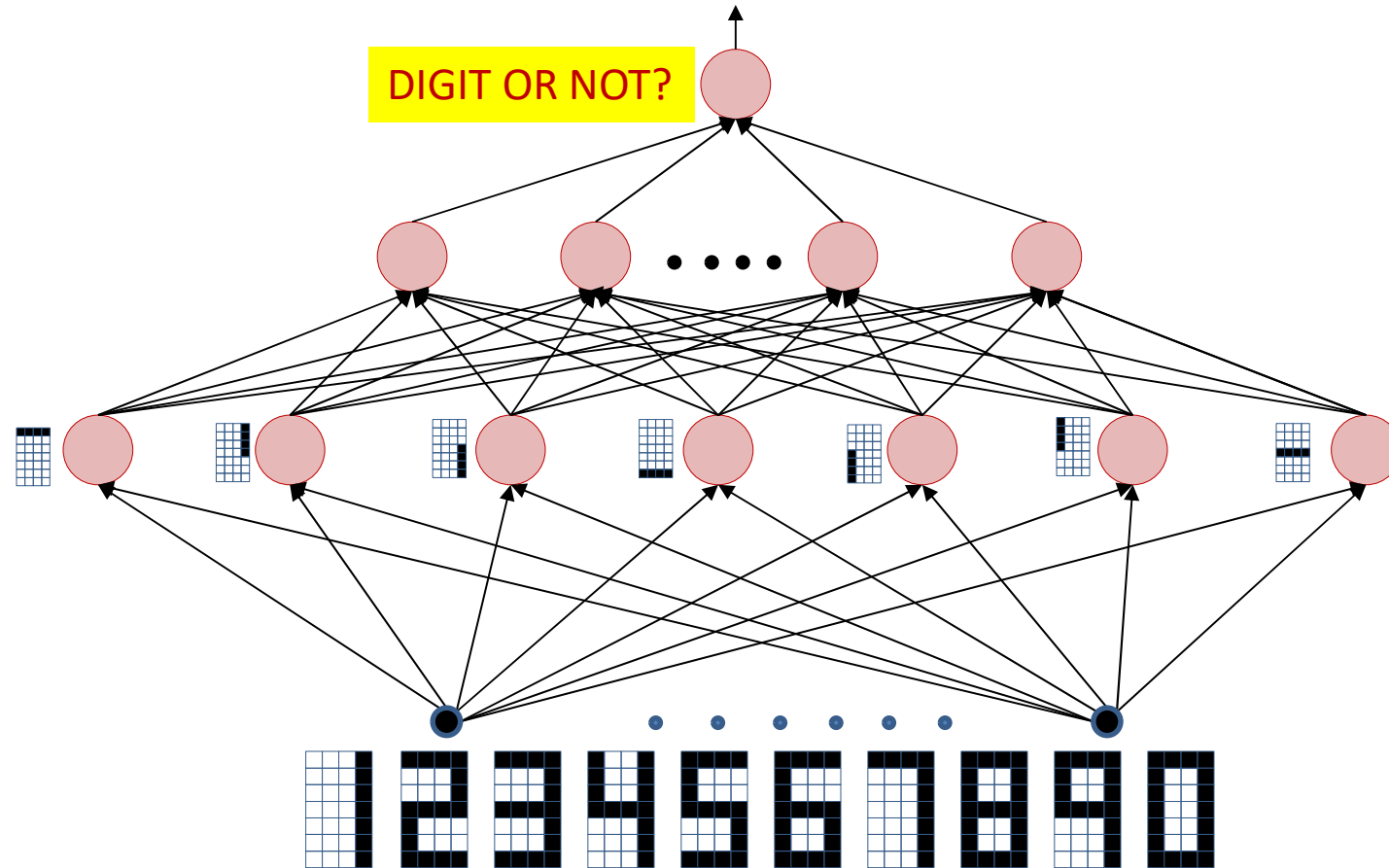


Correlation = 0.82



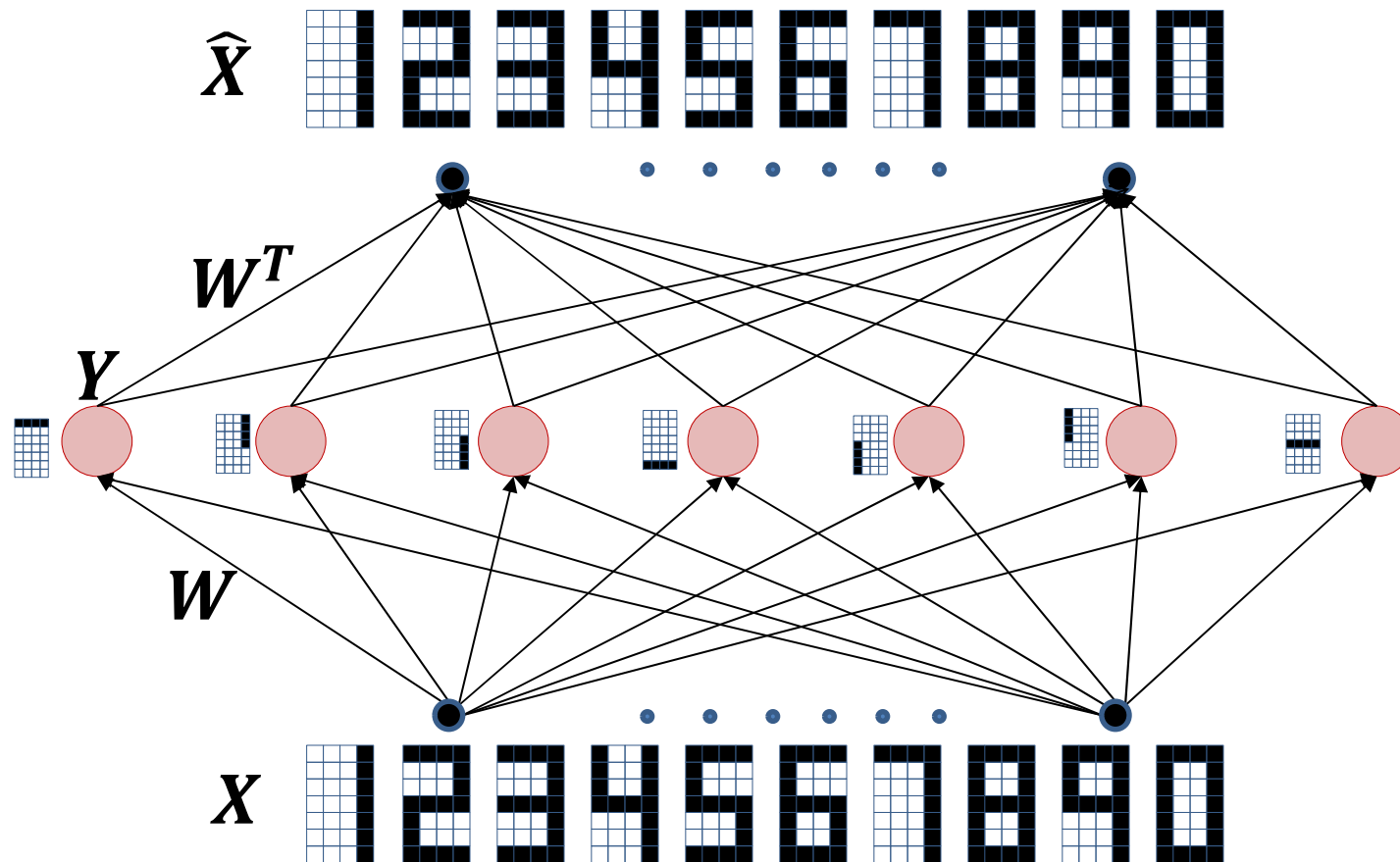
- If the *correlation* between the weight pattern and the inputs exceeds a threshold, fire
- The perceptron is a *correlation filter*!

Recall: MLP features



- The lowest layers of a network detect significant features in the signal
- **The signal could be (partially) reconstructed using these features**
 - Will retain all the significant components of the signal

Making it explicit



- The signal could be (partially) reconstructed using these features
 - Will retain all the significant components of the signal
- Simply *recompose* the detected features
 - Will this work?

Making it explicit

\hat{X} 1 2 3 4 5 6 7 8 9 0

W^T

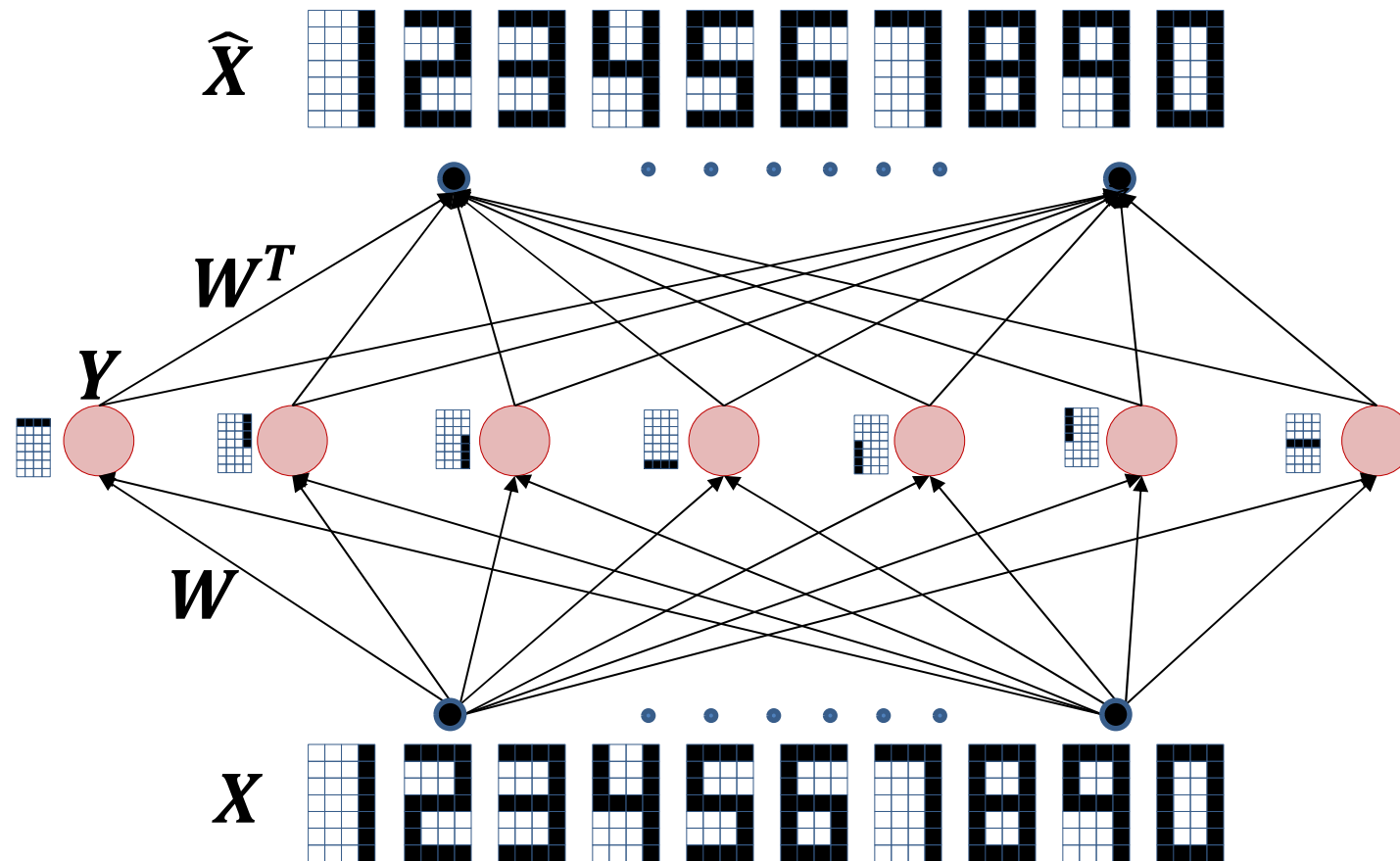
V

X 1 2 3 4 5 6 7 8 9 0

Not in this problem.
The network is optimized to recognize digits
Will only retain distinctly digit-like or obviously not-digit like features
Rest are irrelevant and will be lost

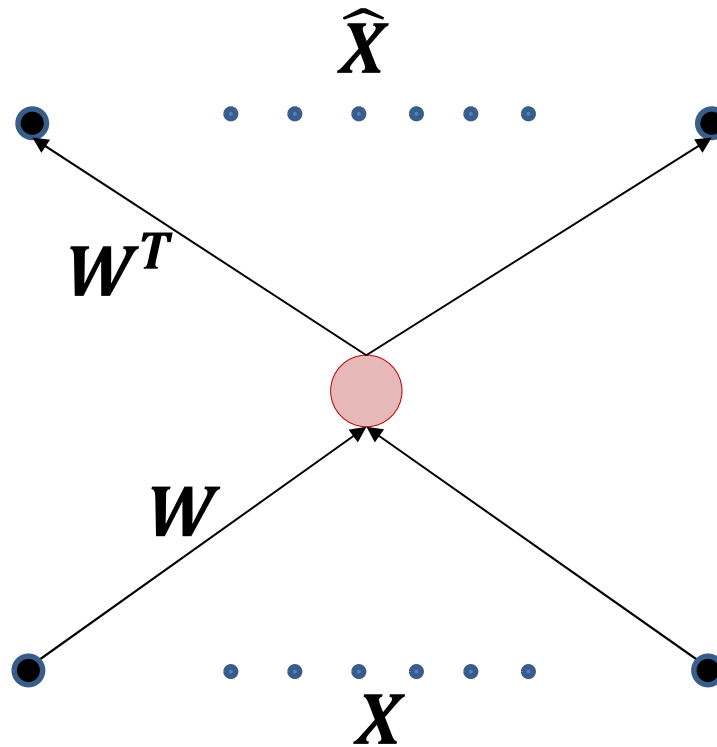
- The signal could be (partially) reconstructed using these features
 - Will retain all the significant components of the signal
- Simply *recompose* the detected features
 - Will this work?

Making it explicit: an autoencoder



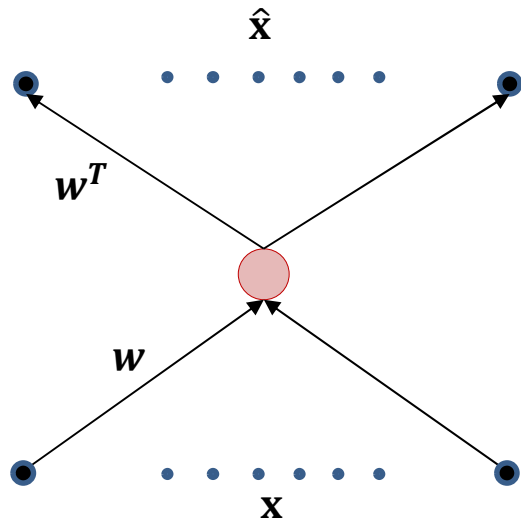
- A neural network can be trained to predict the input itself
- This is an *autoencoder*
- An *encoder* learns to detect all the most significant patterns in the signals
- A *decoder* recomposes the signal from the patterns

The Simplest Autencoder



- A single hidden unit
- Hidden unit has *linear* activation
- What will this learn?

The Simplest Autencoder



Training: Learning W by minimizing L2 divergence

$$\hat{x} = w^T w x$$

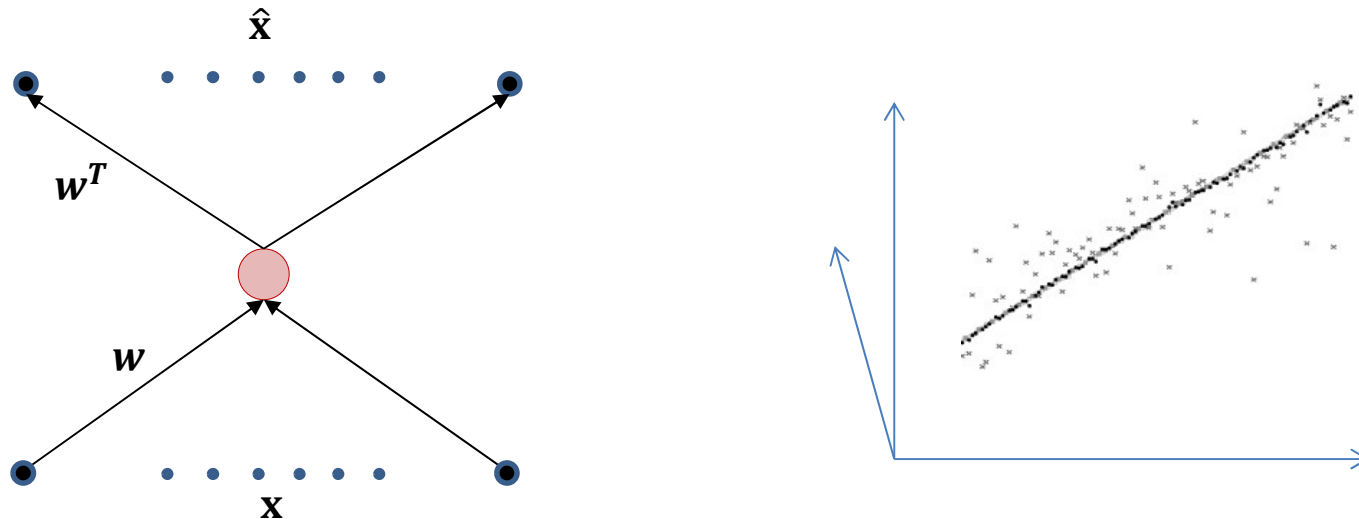
$$\text{div}(\hat{x}, x) = \|x - \hat{x}\|^2 = \|x - w^T w x\|^2$$

$$\hat{W} = \underset{W}{\operatorname{argmin}} E[\text{div}(\hat{x}, x)]$$

$$\hat{W} = \underset{W}{\operatorname{argmin}} E[\|x - w^T w x\|^2]$$

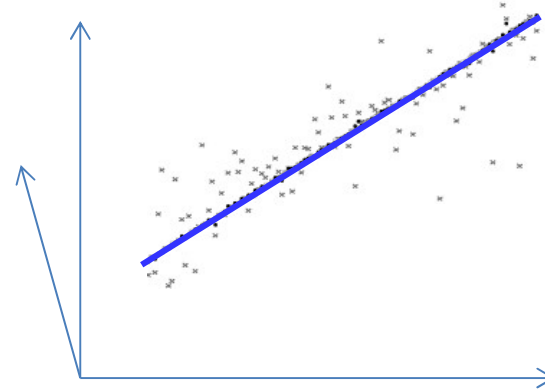
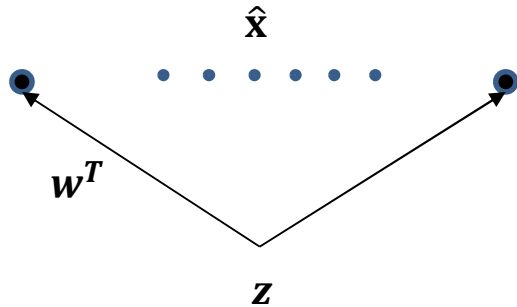
- This is just PCA!

The Simplest Autoencoder



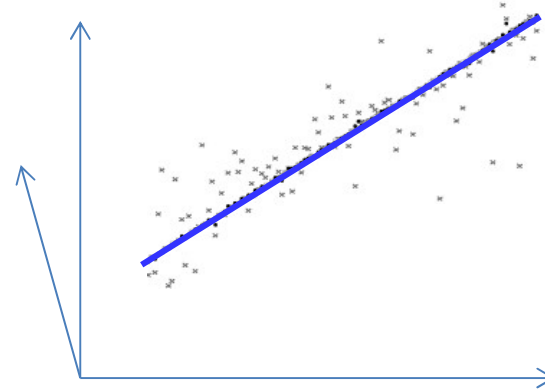
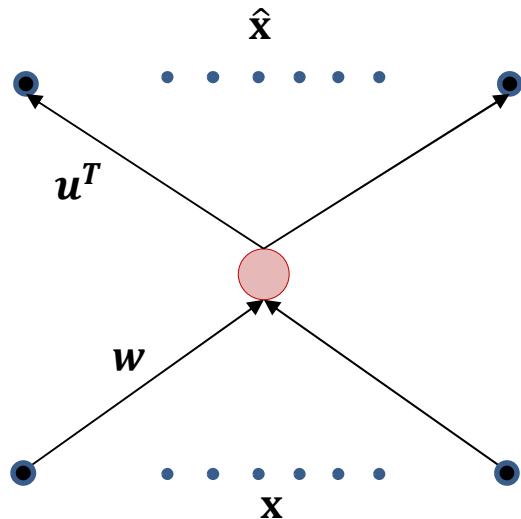
- The autoencoder finds the direction of maximum energy
 - Variance if the input is a zero-mean RV
- All input vectors are mapped onto a point on the principal axis

The Simplest Autencoder



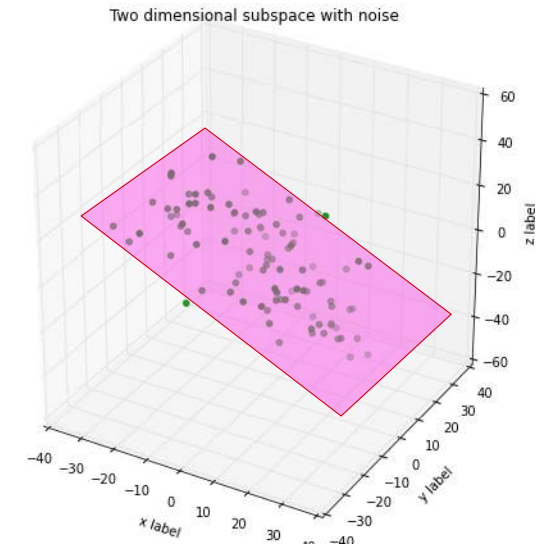
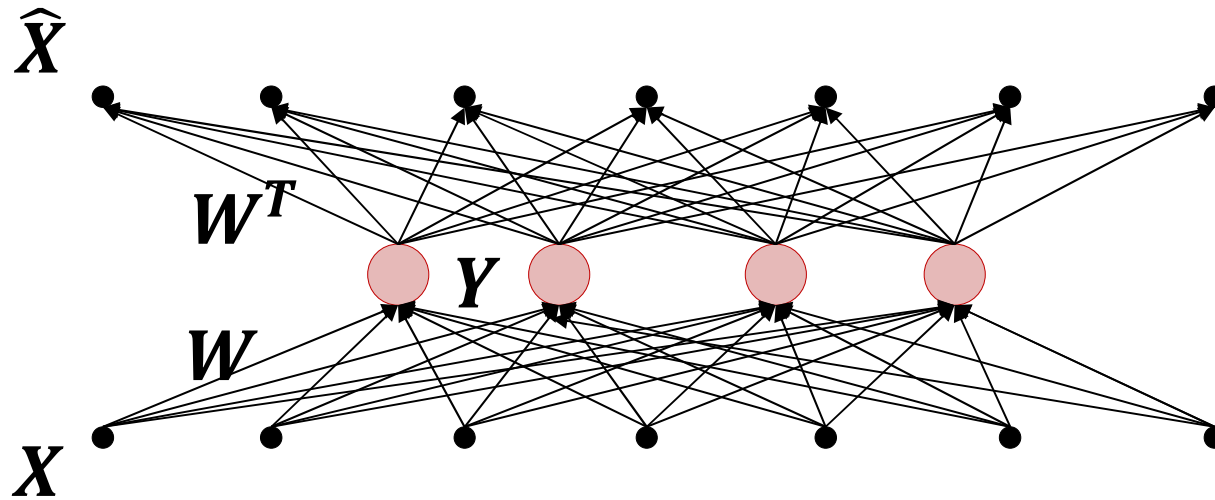
- Simply varying the hidden representation will result in an output that lies along the major axis

The Simplest Autencoder



- Simply varying the hidden representation will result in an output that lies along the major axis
- This will happen even if the learned output weight is separate from the input weight
 - The minimum-error direction *is* the principal eigen vector

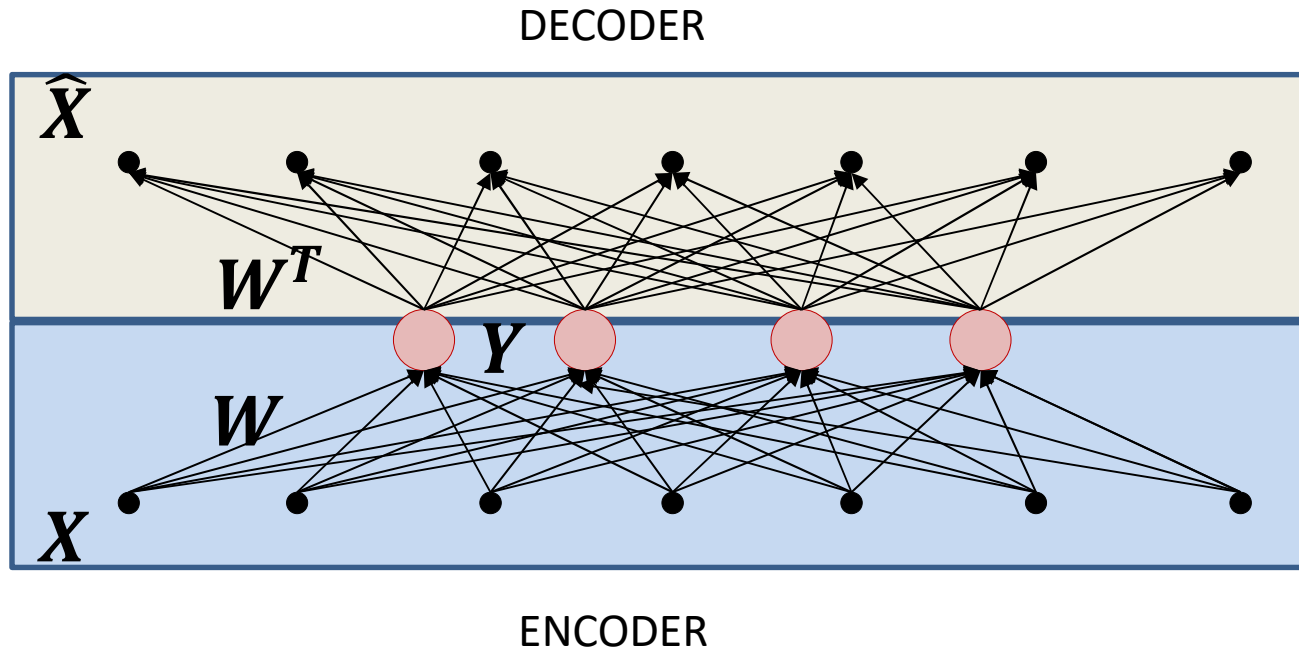
For more detailed AEs without a non-linearity



$$\boxed{Y = WX} \quad \boxed{\hat{X} = W^T Y} \quad E = \|X - W^T W X\|^2 \quad \text{Find } W \text{ to minimize Avg}[E]$$

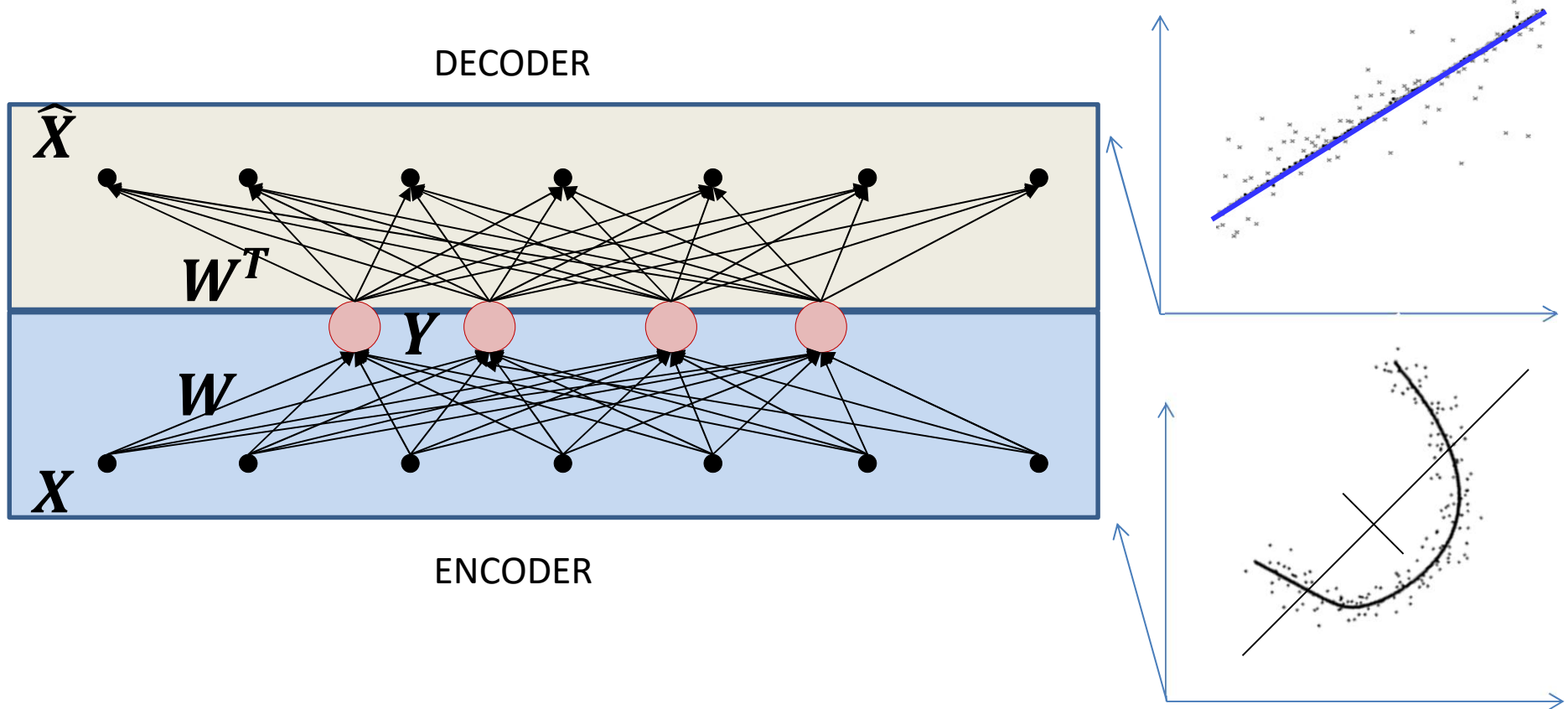
- This is still just PCA
 - The output of the hidden layer will be in the principal subspace
 - Even if the recomposition weights are different from the “analysis” weights

Terminology



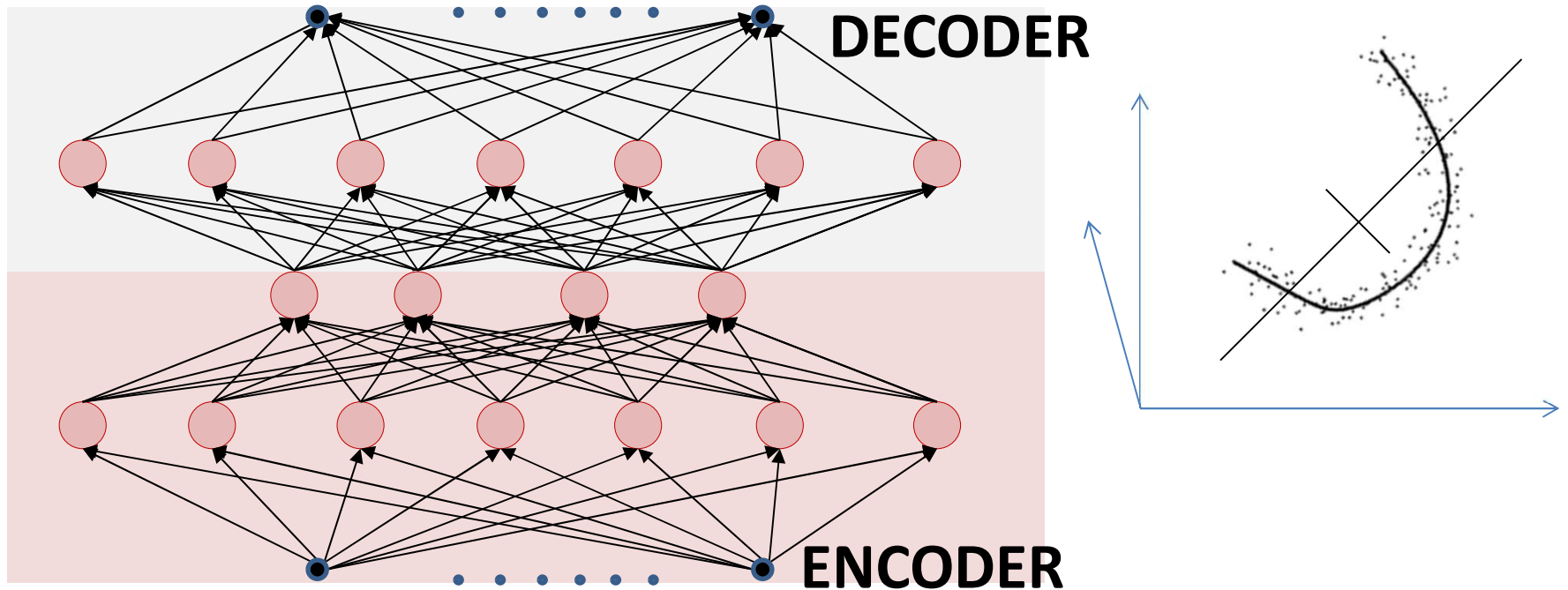
- Terminology:
 - **Encoder:** The “Analysis” net which computes the hidden representation
 - **Decoder:** The “Synthesis” which recomposes the data from the hidden representation

Introducing *nonlinearity*



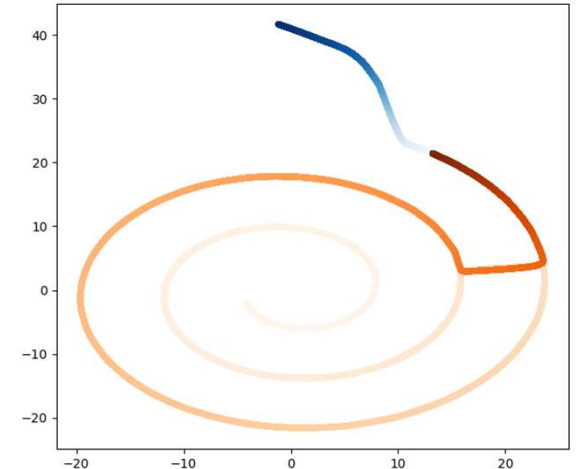
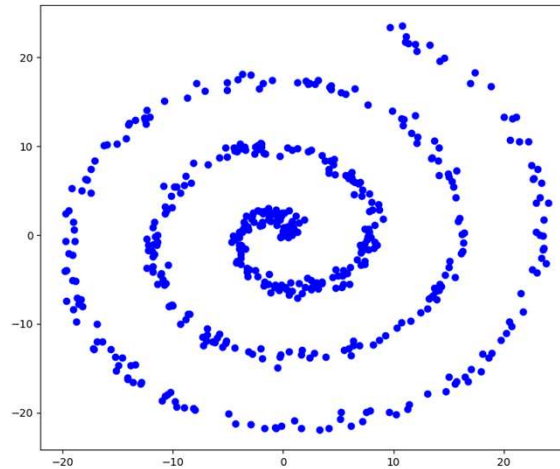
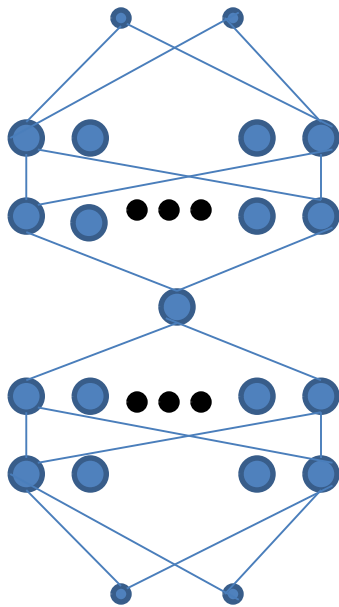
- When the hidden layer has a *linear* activation the decoder represents the best *linear* manifold to fit the data
 - Varying the hidden value will move along this linear manifold
- **When the hidden layer has non-linear activation, the net performs *nonlinear PCA***
 - The decoder represents the best non-linear manifold to fit the data
 - Varying the hidden value will move along this non-linear manifold

The AE



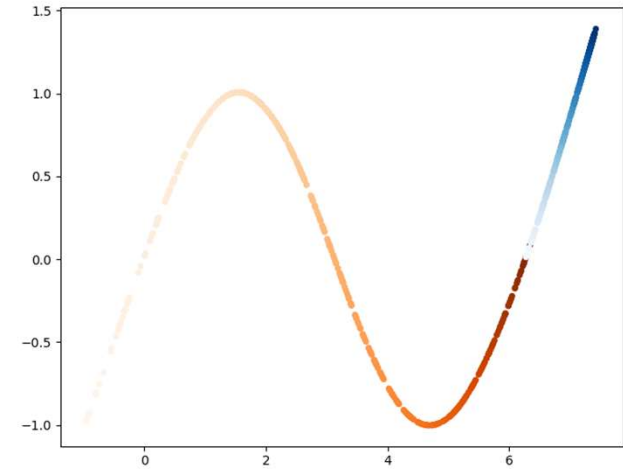
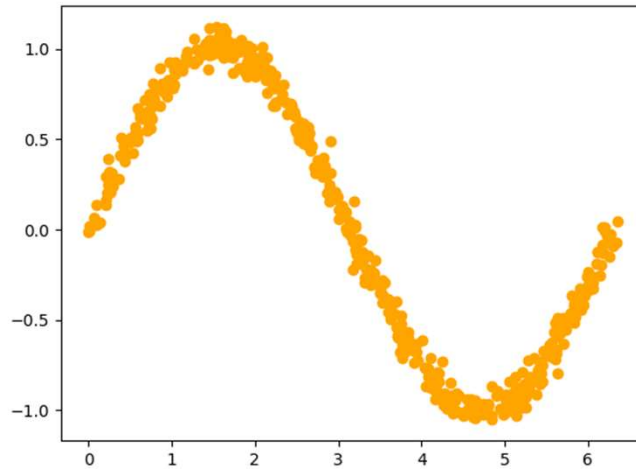
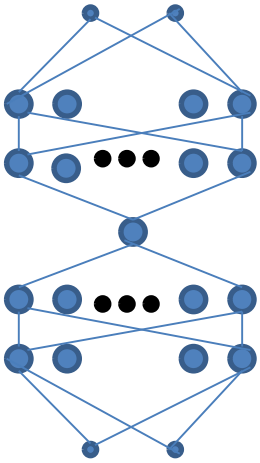
- With non-linearity
 - “Non linear” PCA
 - Deeper networks can capture more complicated manifolds
 - “Deep” autoencoders

Some examples



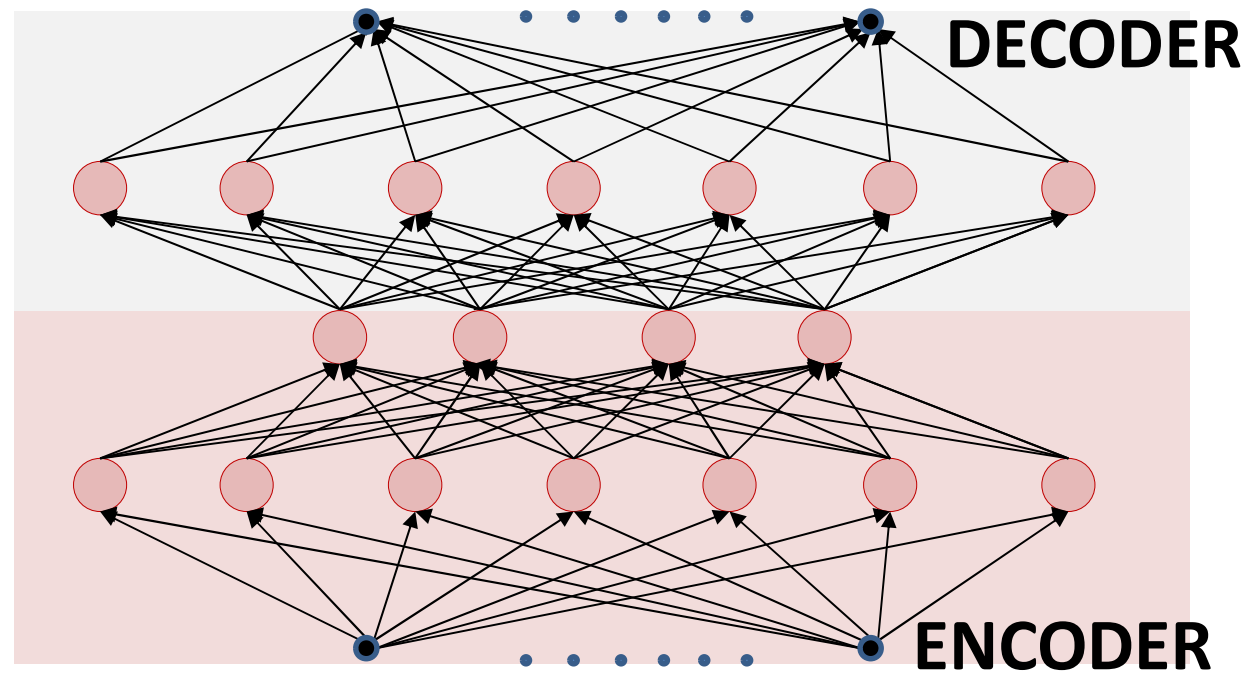
- 2-D input
- Encoder and decoder have 2 hidden layers of 100 neurons, but hidden representation is unidimensional
- Extending the hidden “z” value beyond the values seen in training does not continue along a helix

Some examples



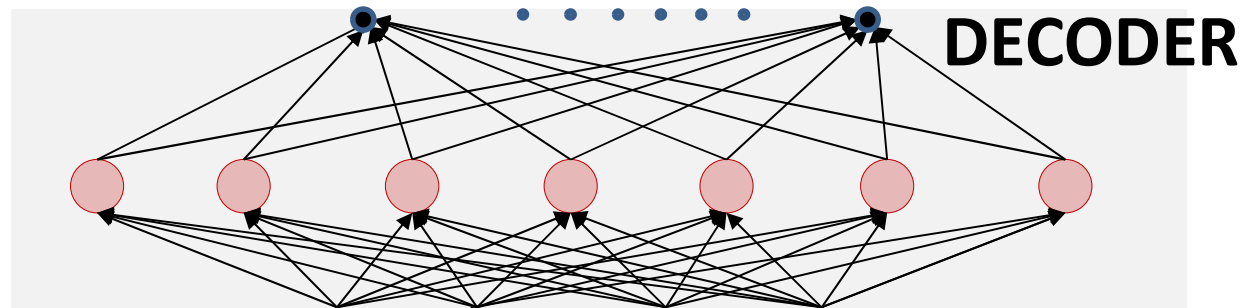
- The model is specific to the training data..
 - Varying the hidden layer value only generates data along the learned manifold
 - *Any input* will result in an output along the learned manifold
 - But may not generalize beyond the manifold

The AE



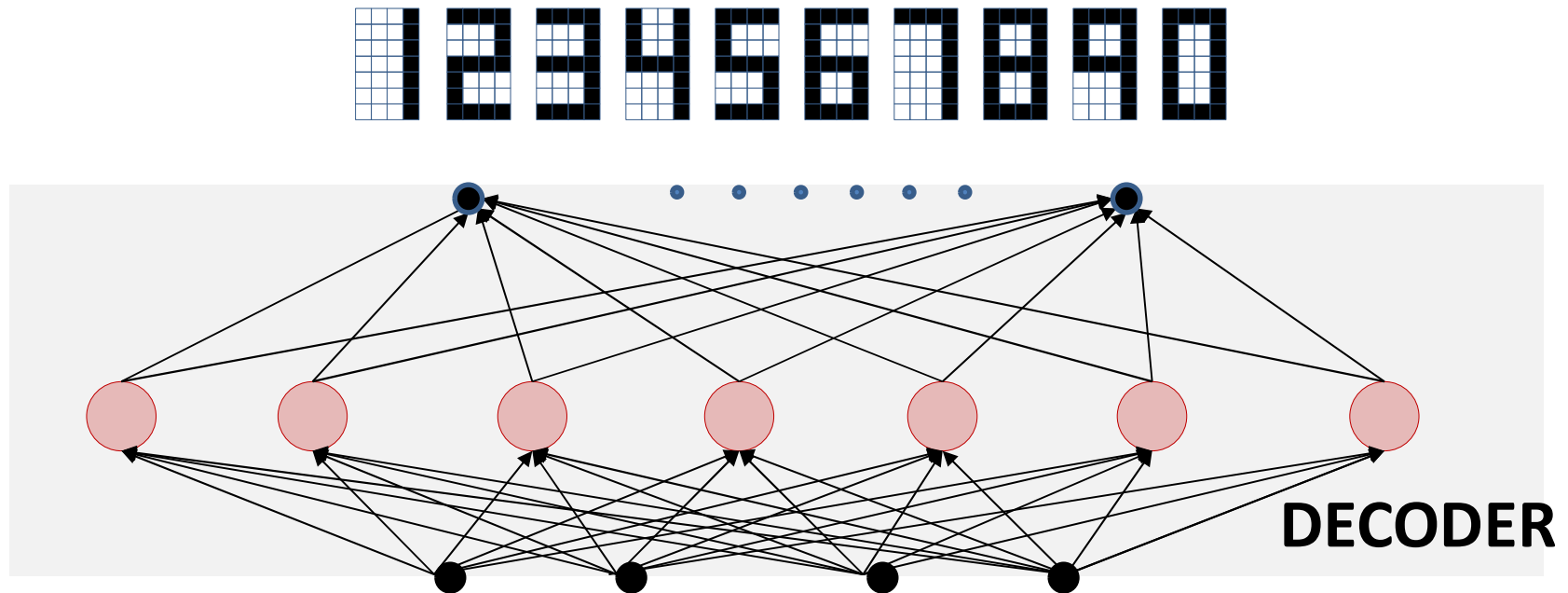
- When the hidden representation is of lower dimensionality than the input, often called a “**bottleneck**” network
 - Nonlinear PCA
 - Learns the manifold for the data
 - If properly trained

The AE



- The decoder can only generate data on the manifold that the training data lie on
- This also makes it an excellent “generator” of the distribution of the training data
 - Any values applied to the (hidden) input to the decoder will produce data similar to the training data

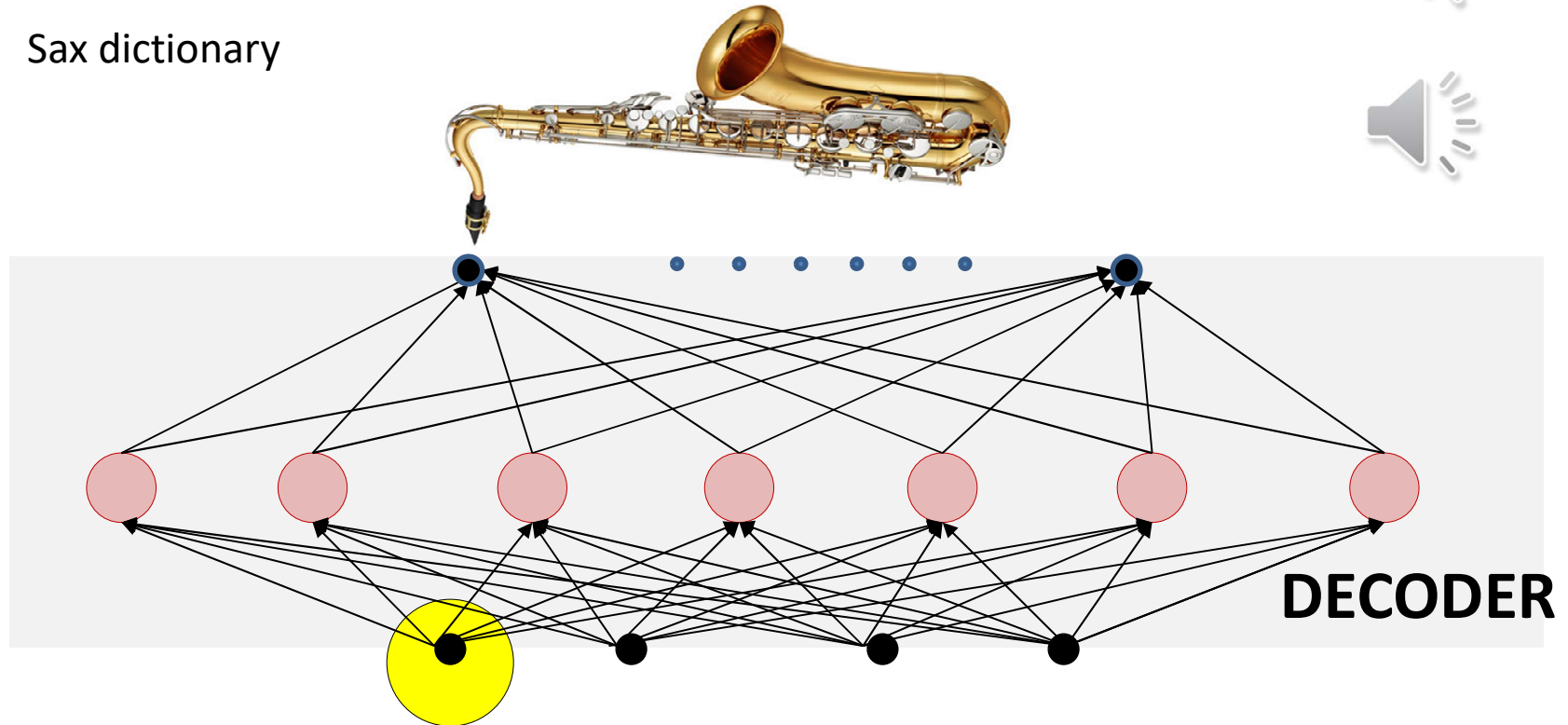
The Decoder:



- The decoder represents a source-specific generative *dictionary*
- Exciting it will produce typical data from the source!

The Decoder:

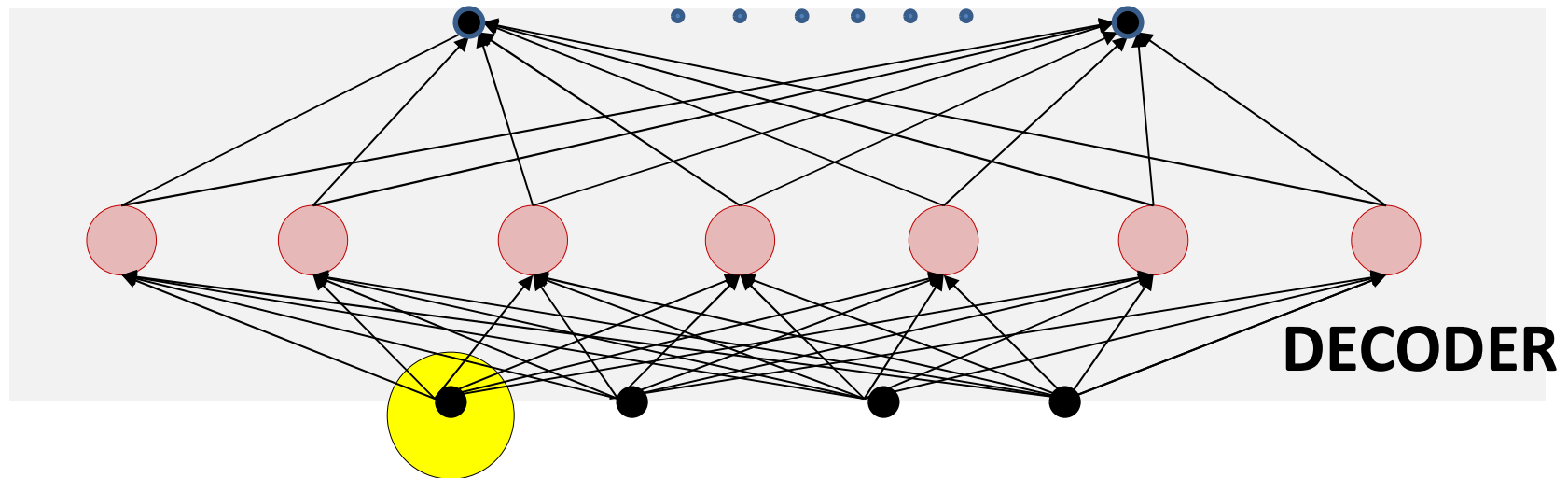
Sax dictionary



- The decoder represents a source-specific generative *dictionary*
- Exciting it will produce typical data from the source!

The Decoder:

Clarinet dictionary

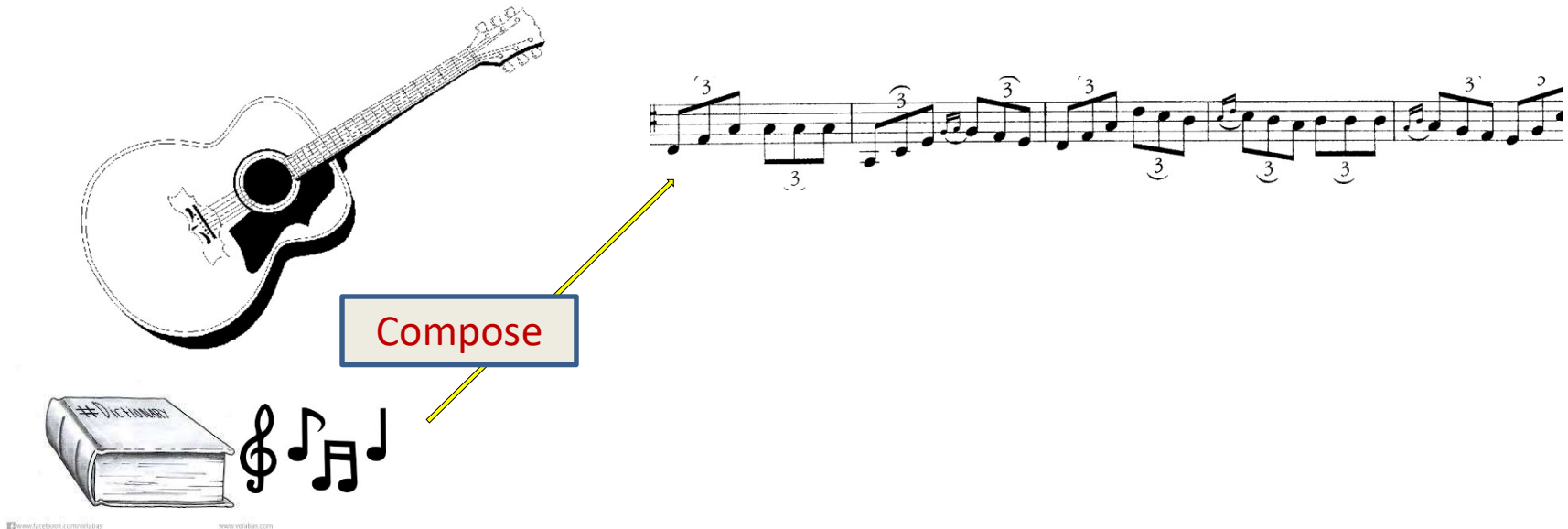


- The decoder represents a source-specific generative *dictionary*
- Exciting it will produce typical data from the source!

A cute application..

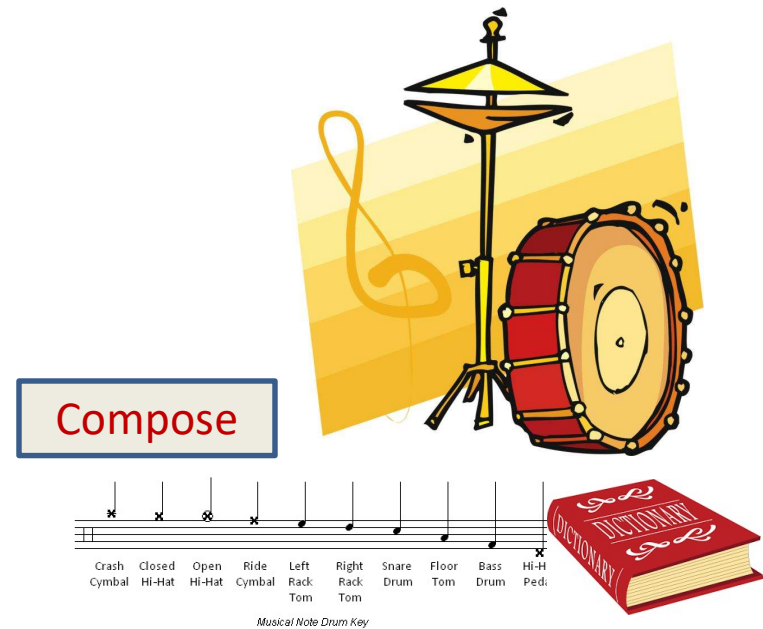
- Signal separation...
- Given a mixed sound from multiple sources, separate out the sources

Dictionary-based techniques



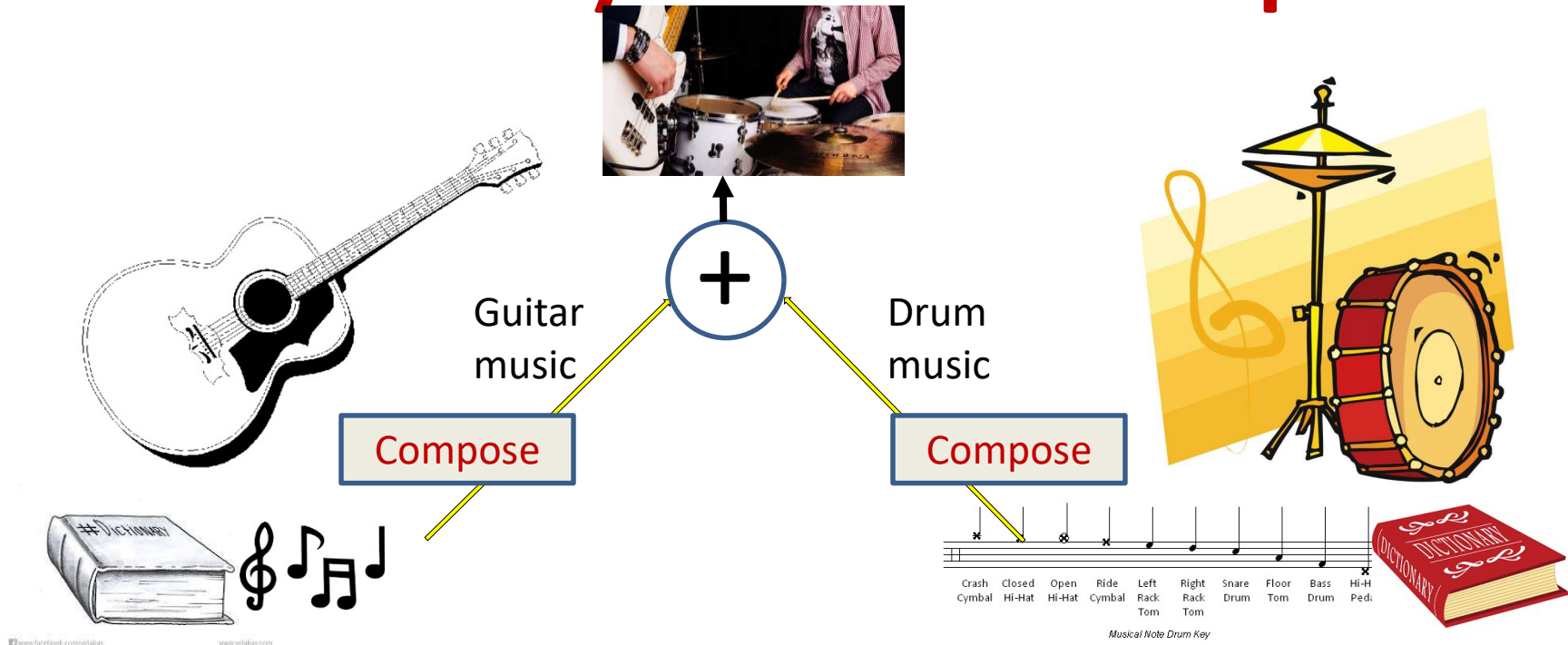
- Basic idea: Learn a dictionary of “building blocks” for each sound source
- All signals by the source are composed from entries from the dictionary for the source

Dictionary-based techniques



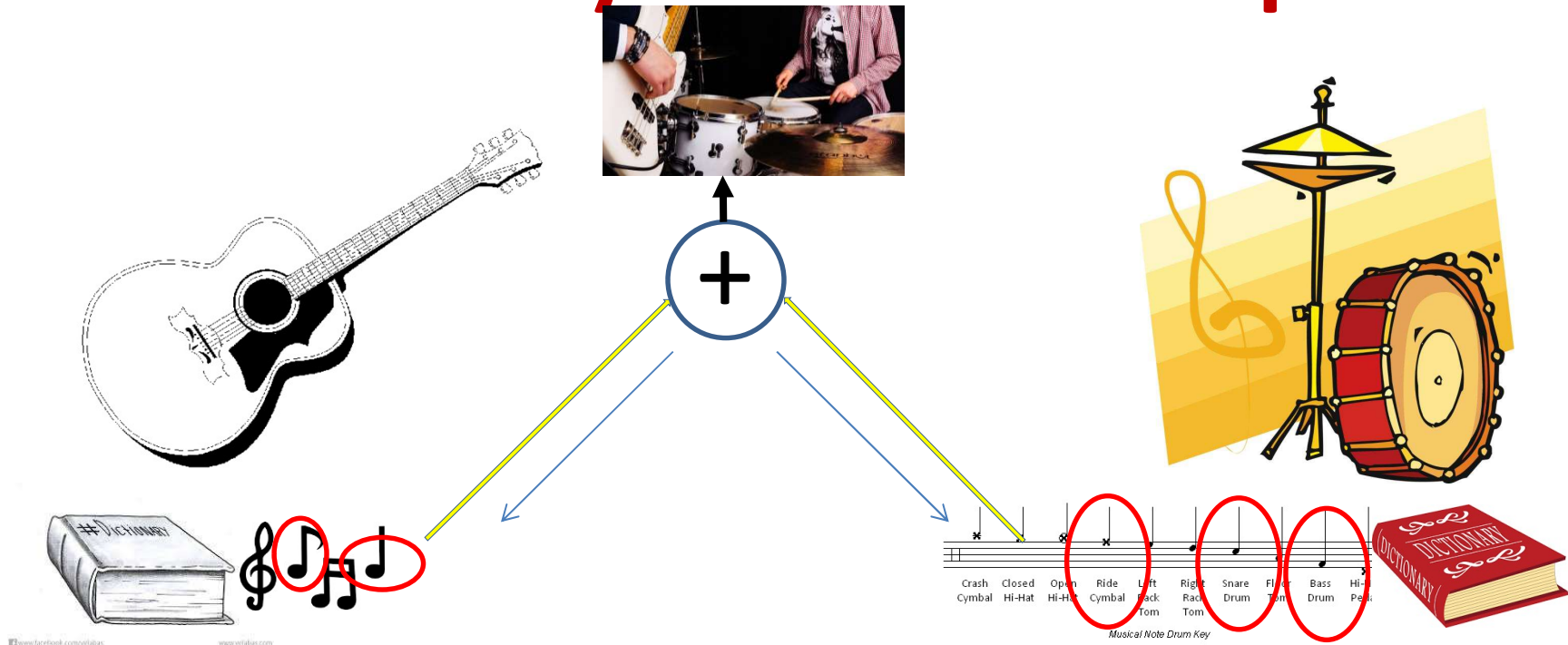
- Learn a similar dictionary for all sources expected in the signal

Dictionary-based techniques



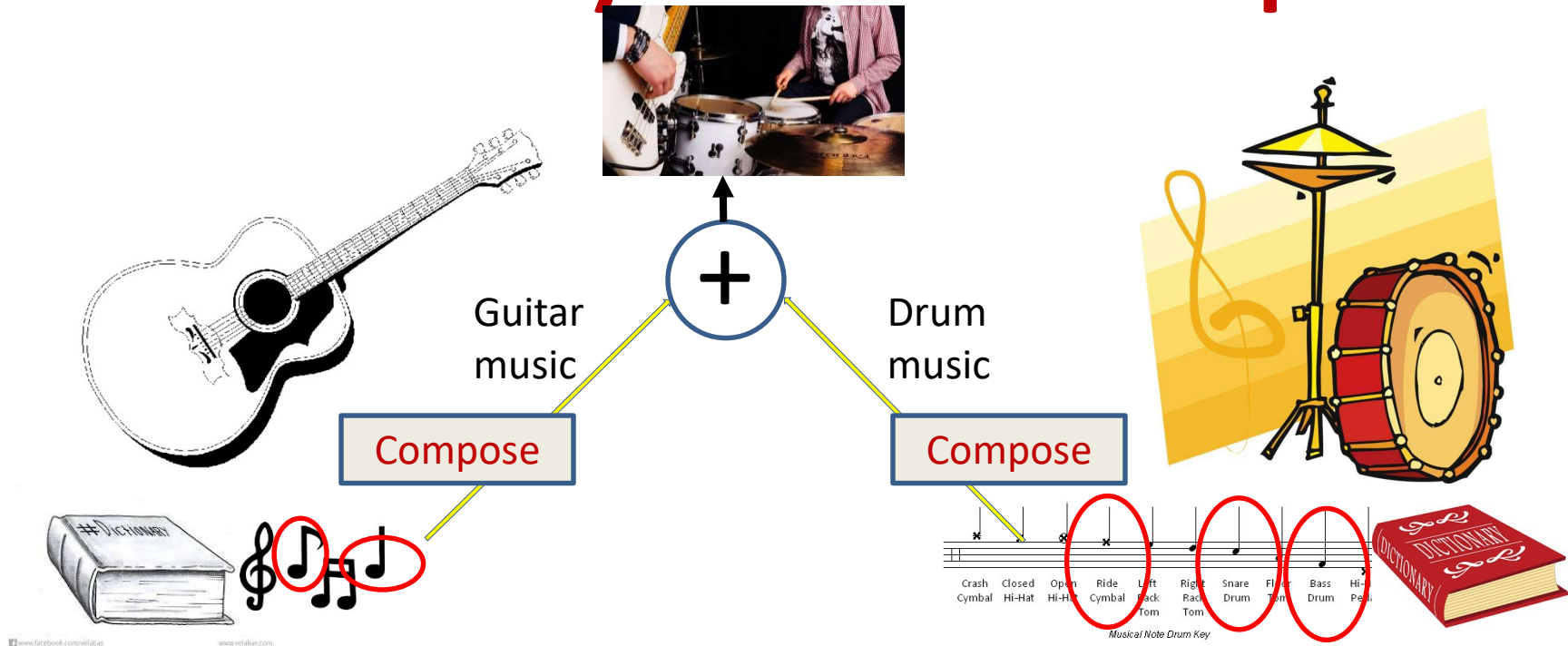
- A mixed signal is the linear combination of signals from the individual sources
 - Which are in turn composed of entries from its dictionary

Dictionary-based techniques



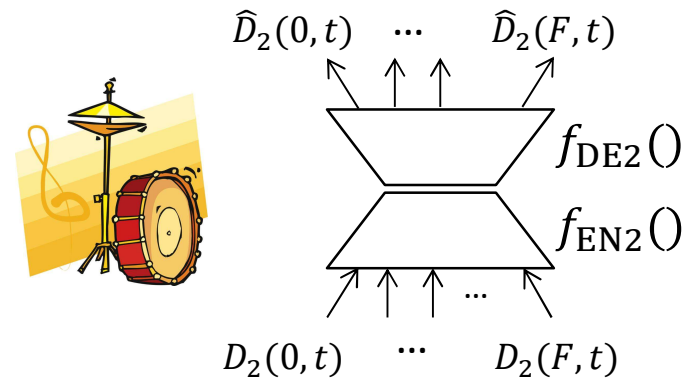
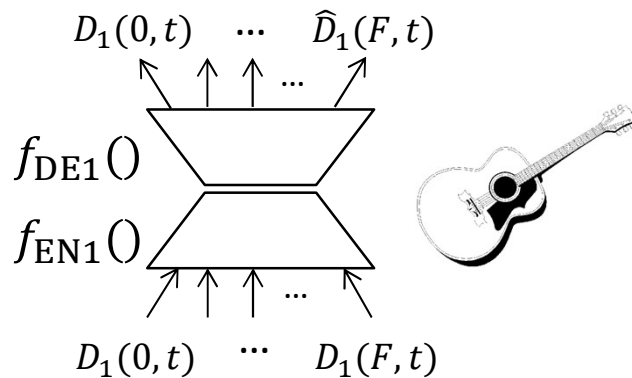
- Separation: Identify the combination of entries from both dictionaries that compose the mixed signal

Dictionary-based techniques



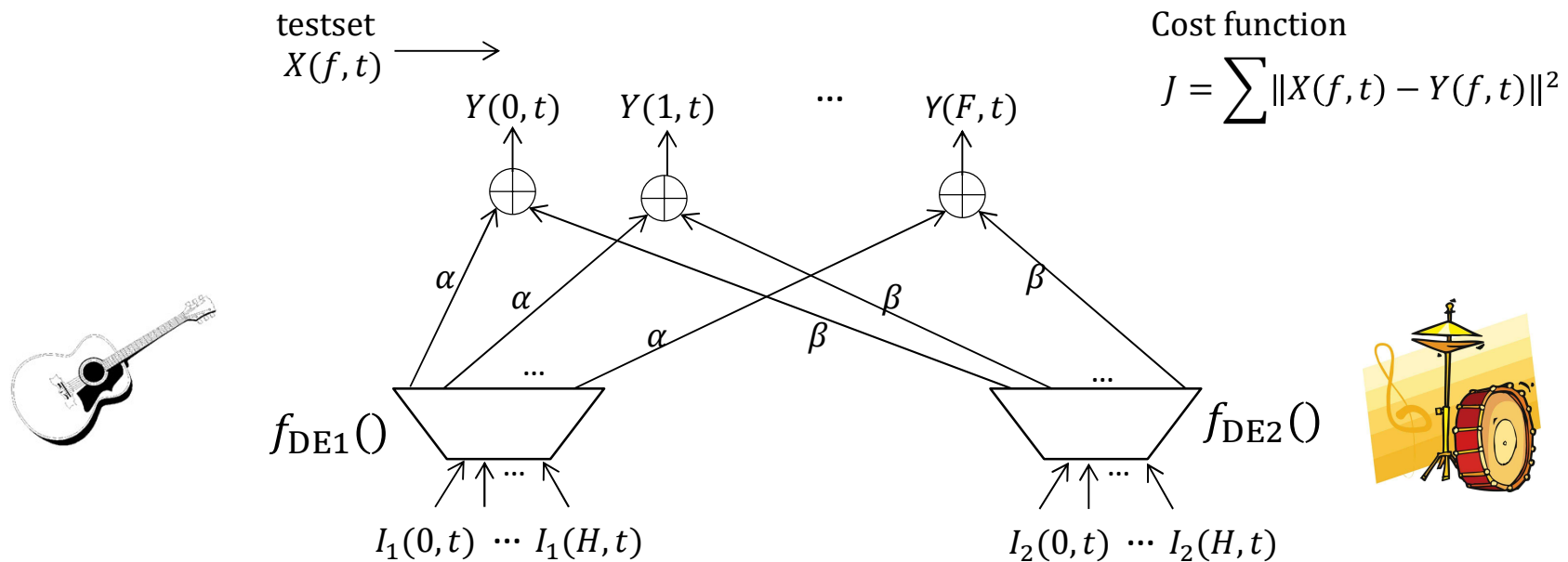
- Separation: Identify the combination of entries from both dictionaries that compose the mixed signal
 - The composition from the identified dictionary entries gives you the separated signals

Learning Dictionaries



- Autoencoder dictionaries for each source
 - Operating on (magnitude) spectrograms
- For a well-trained network, the “decoder” dictionary is highly specialized to creating sounds for that source

Model for mixed signal



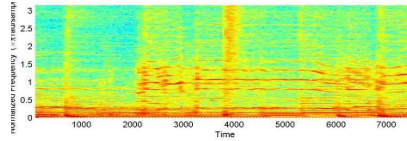
Estimate $I_1()$ and $I_2()$ to minimize cost function $J()$

- The sum of the outputs of both neural dictionaries
 - For some unknown input

Separation

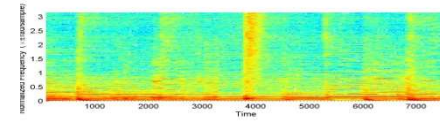
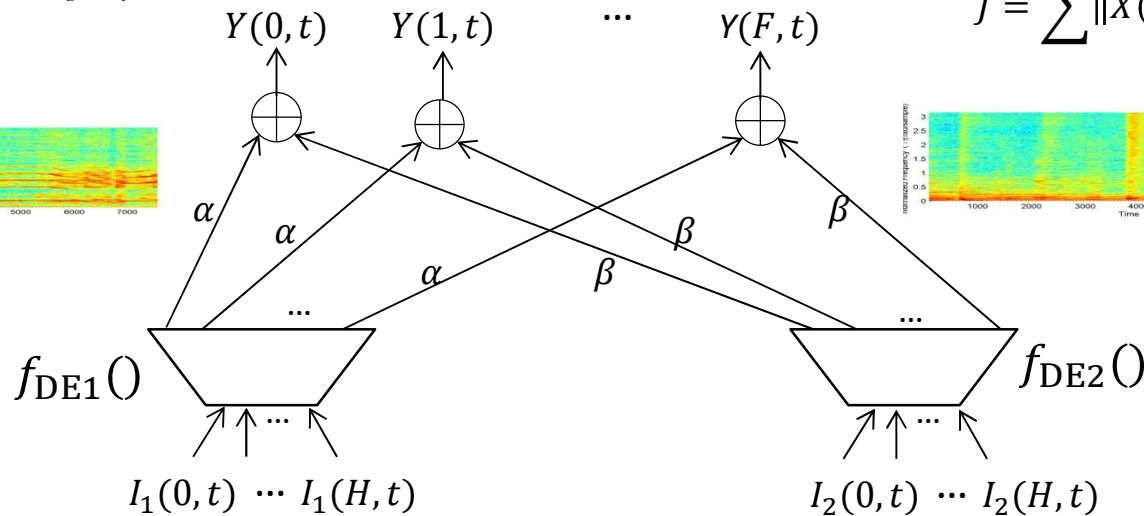
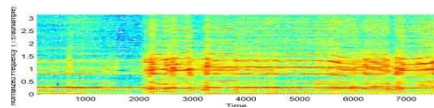
Test Process

testset
 $X(f, t)$



Cost function

$$J = \sum \|X(f, t) - Y(f, t)\|^2$$



Estimate $I_1()$ and $I_2()$ to minimize cost function $J()$

- Given mixed signal and source dictionaries, find excitation that best recreates mixed signal
 - Simple backpropagation
- Intermediate results are separated signals

Example Results

Mixture



Separated



Separated



Original



Original

5-layer dictionary, 600 units wide

- Separating music

Story for the day

- Classification networks learn to predict the *a posteriori* probabilities of classes
 - The network until the final layer is a feature extractor that converts the input data to be (almost) linearly separable
 - The final layer is a classifier/predictor that operates on linearly separable data
- Neural networks can be used to perform linear or non-linear PCA
 - “Autoencoders”
 - Can also be used to compose constructive dictionaries for data
 - Which, in turn can be used to model data distributions