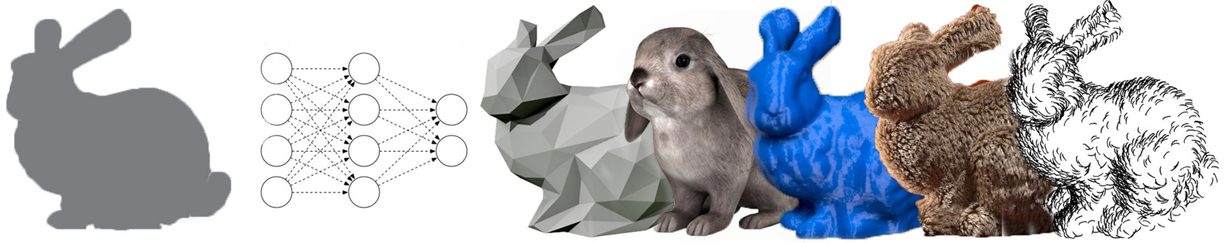


Machine Learning Basics

Niloy Mitra
UCL

Iasonas Kokkinos
UCL

Federico Monti
USI Lugano

Emanuele Rodolà
La Sapienza

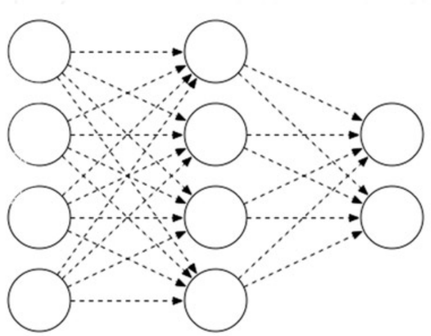
Michael Bronstein
Imperial College
USI Lugano

Or Litany
Stanford University
Facebook

Leonidas Guibas
Stanford University

http://geometry.cs.ucl.ac.uk/dl_for_CG/

Course Information (slides/code/comments)



http://geometry.cs.ucl.ac.uk/dl_for_CG/

Scan me

Timetable

			Niloy	Federico	Iasonas	Emanuele
Theory/Basics	Introduction	9:00	X	X	X	X
	Machine Learning Basics	~ 9:05	X			
	Neural Network Basics	~ 9:35		X		
	Alternatives to Direct Supervision (GANs)	~11:00			X	
State of the Art	Image Domain	~11:45			X	
	3D Domains (extrinsic)	~13:30	X			
	3D Domains (intrinsic)	~ 14:15				X
	Physics and Animation	~ 16:00	X			
	Discussion	~ 16:45	X	X	X	X

Sessions: A. 9:00-10:30 (**coffee**) B. 11:00-12:30 [**LUNCH**] C. 13:30-15:00 (**coffee**) D. 15:30-17:00

Machine Learning Variants

- **Supervised**
  - Classification
  - Regression
  - Data consolidation
- **Unsupervised**
  - Clustering
  - Dimensionality Reduction
- **Weakly supervised/semi-supervised**

  Some data supervised, some unsupervised
- **Reinforcement learning**

  Supervision: sparse reward for a sequence of decisions

Machine Learning Variants

- **Supervised**
  - **Classification**
  - Regression
  - Data consolidation
- **Unsupervised**
  - Clustering
  - Dimensionality Reduction
- **Weakly supervised/semi-supervised**

  Some data supervised, some unsupervised
- **Reinforcement learning**

  Supervision: sparse reward for a sequence of decisions

Classification Examples

- Digit Recognition

Classification Examples

- Digit Recognition

- Spam Detection

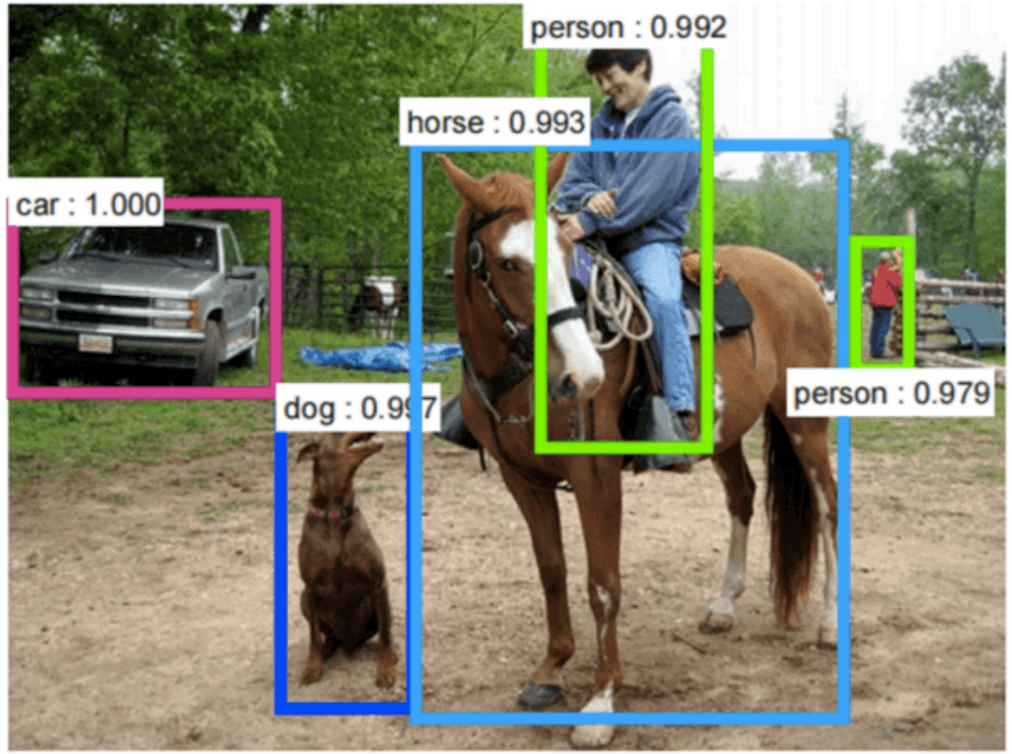
Classification Examples

- Digit Recognition

- Spam Detection

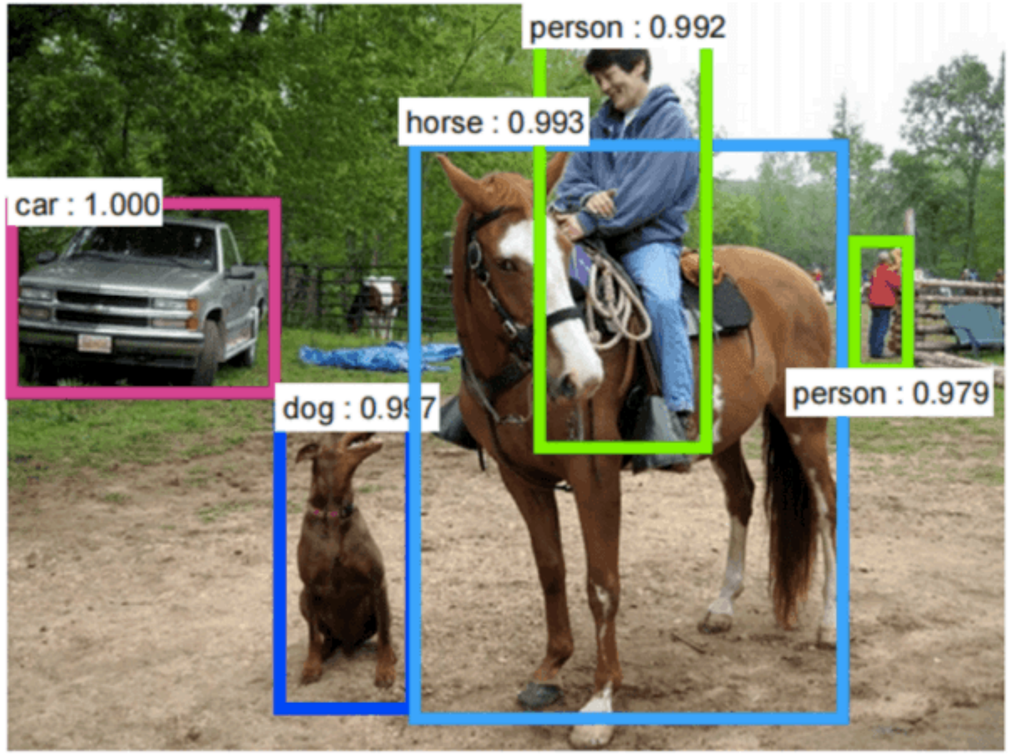
- Face detection

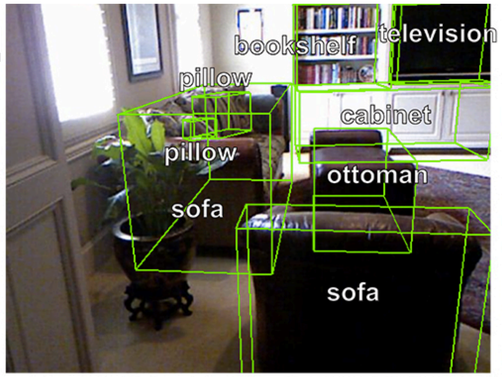
Segmentation + Classification in Real Images

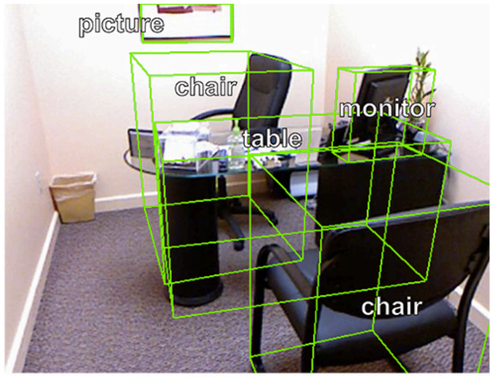


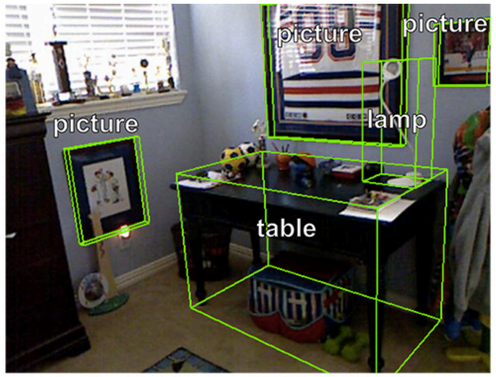
uld we give a loan to this customer?

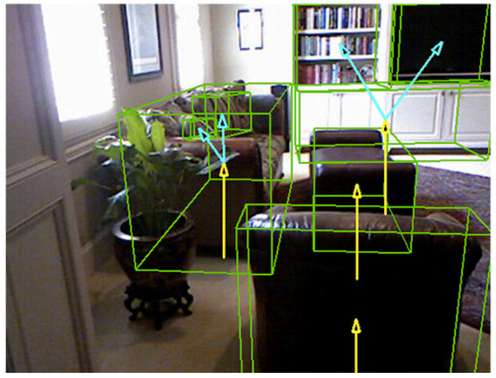
Segmentation + Classification in Real Images

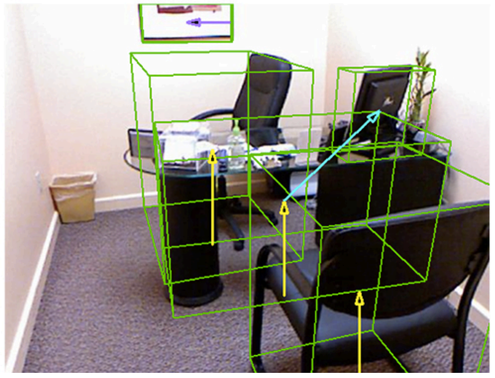


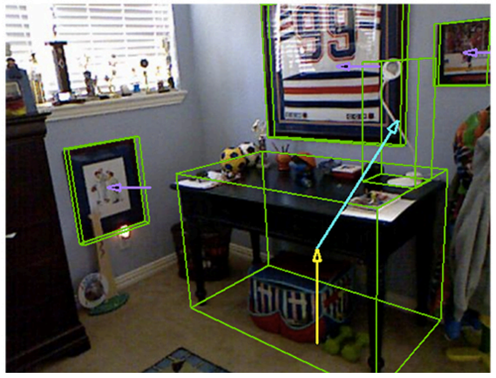




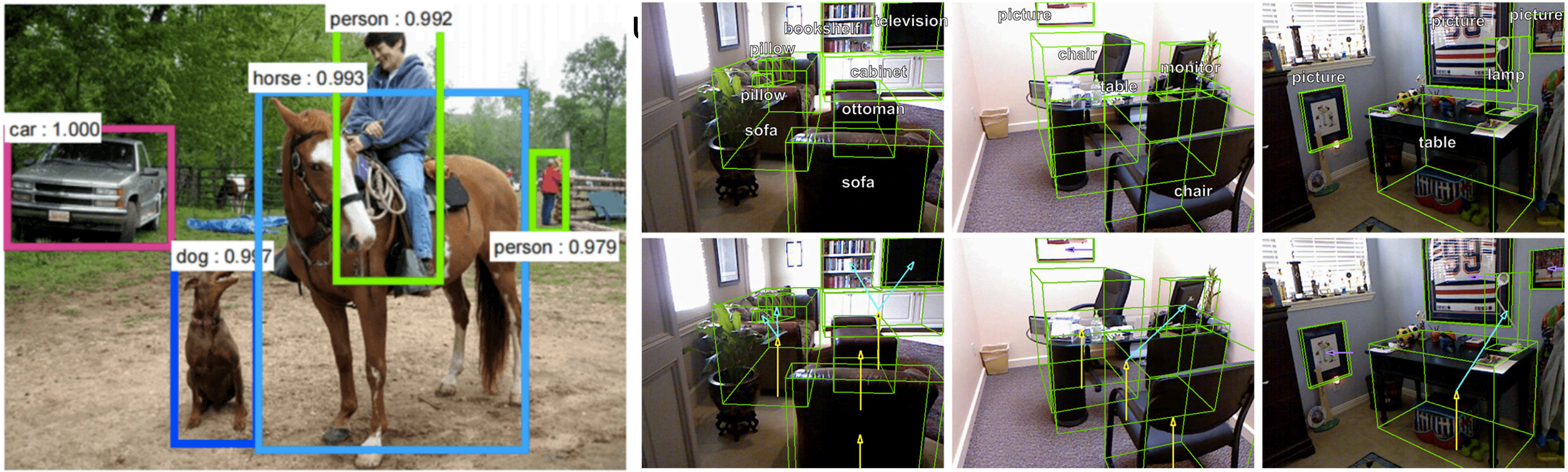








Segmentation + Classification in Real Images



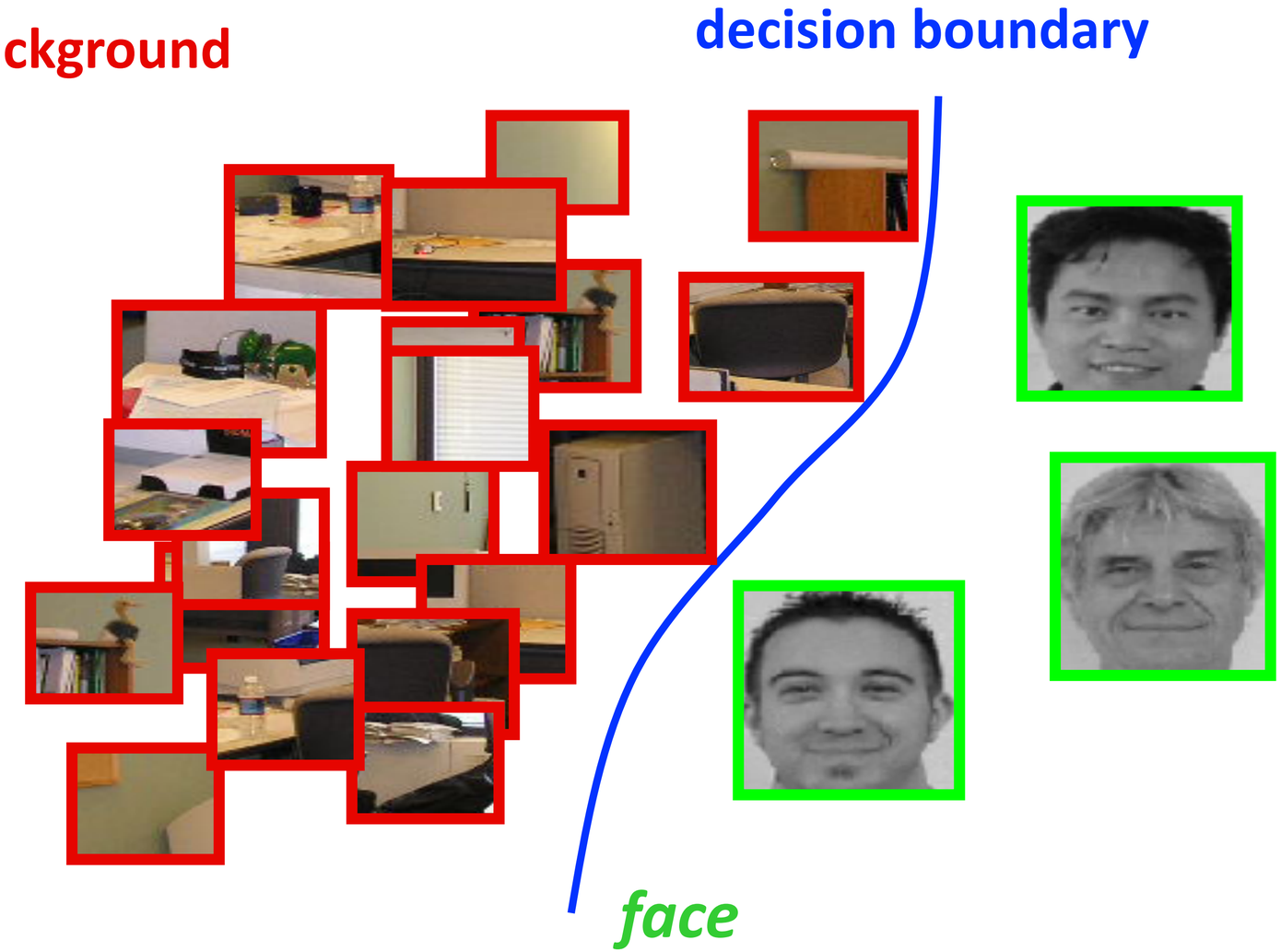
Evaluation measures: Confusion matrix, ROC curve, precision, recall, etc.

`Faceness' Function: Classifier

background

face

`Faceness' Function: Classifier



Machine Learning Variants

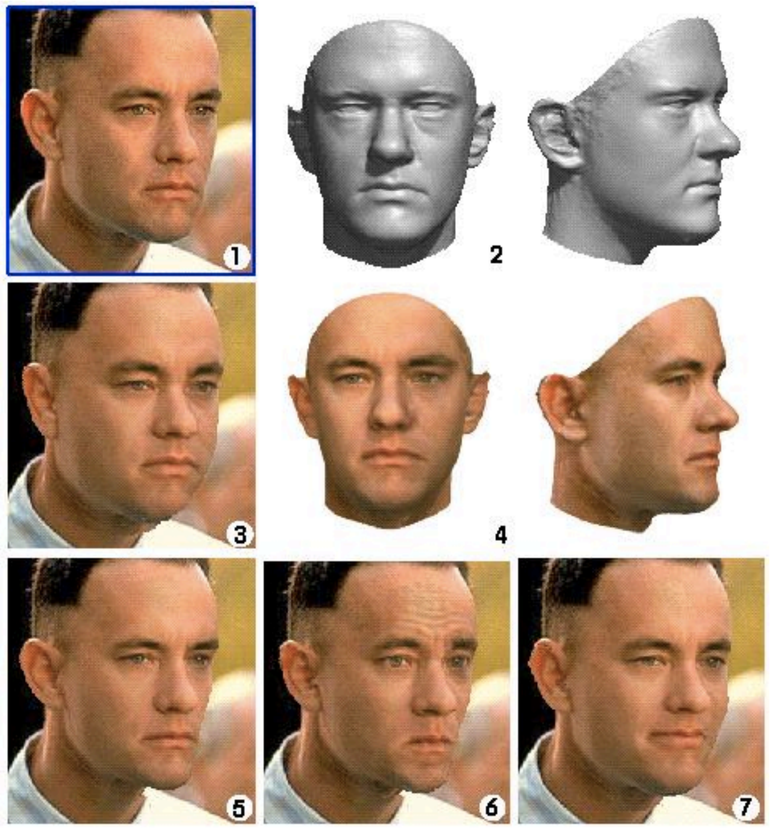
- **Supervised**
  - Classification
  - **Regression**
  - Data consolidation
- **Unsupervised**
  - Clustering
  - Dimensionality Reduction
- **Weakly supervised/semi-supervised**

  Some data supervised, some unsupervised
- **Reinforcement learning**

  Supervision: sparse reward for a sequence of decisions

Human Face/Pose Estimation

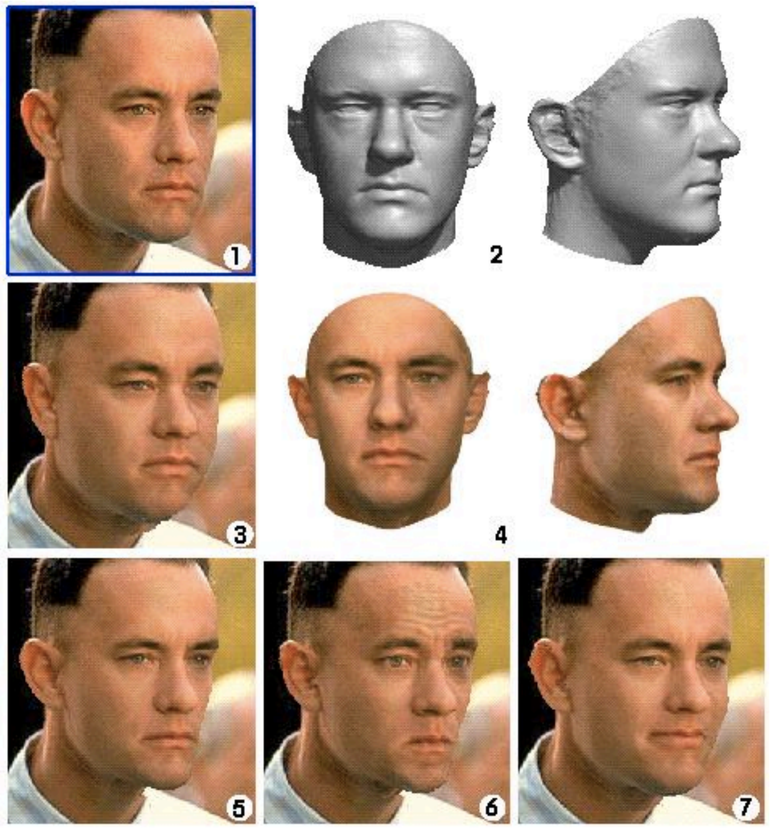
- Human estimation: from image to vector-valued pose estimate



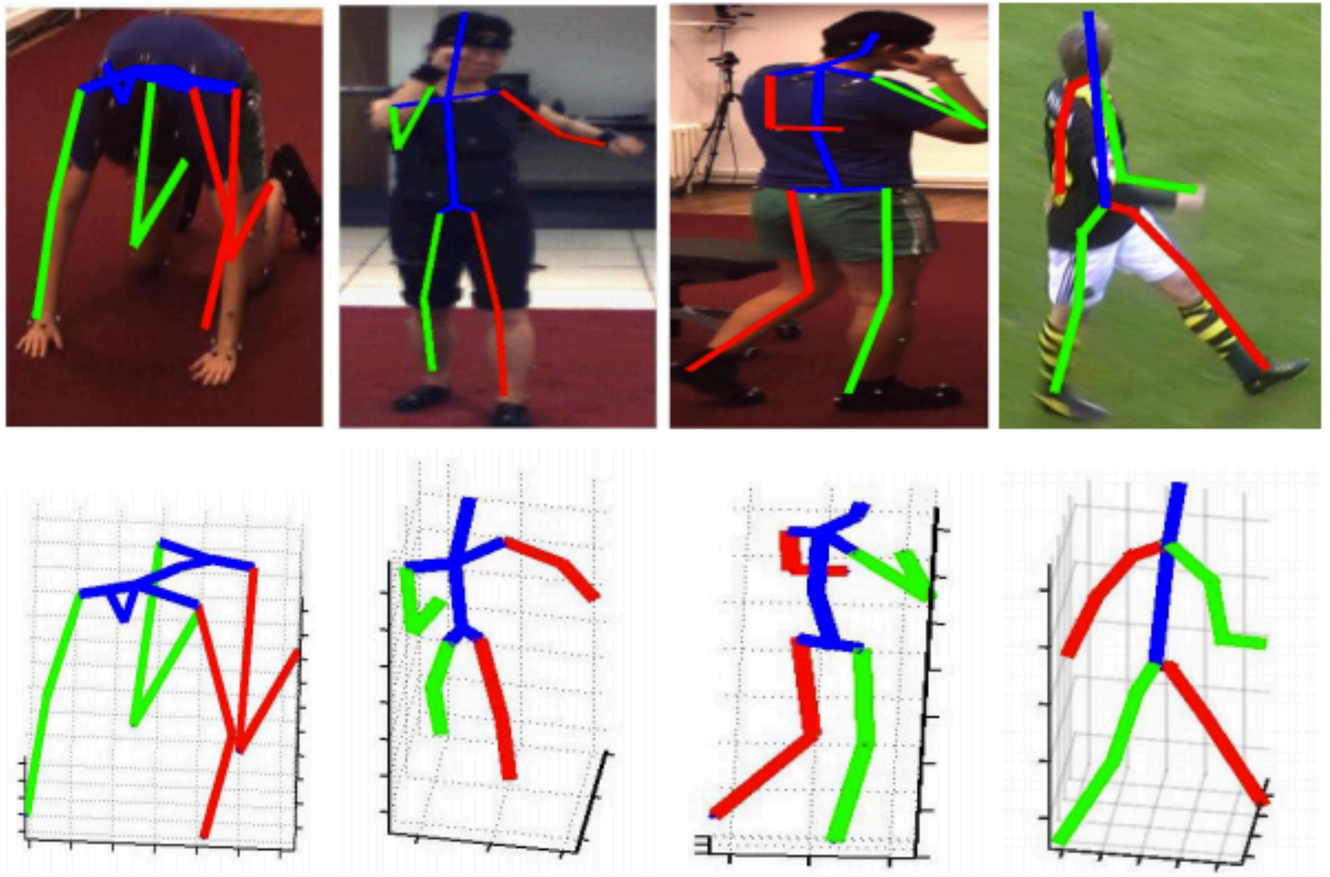
[Blanz and Vetter, Siggraph, 1999]

Human Face/Pose Estimation

- Human estimation: from image to vector-valued pose estimate



[Blanz and Vetter, Siggraph, 1999]



Machine Learning Variants

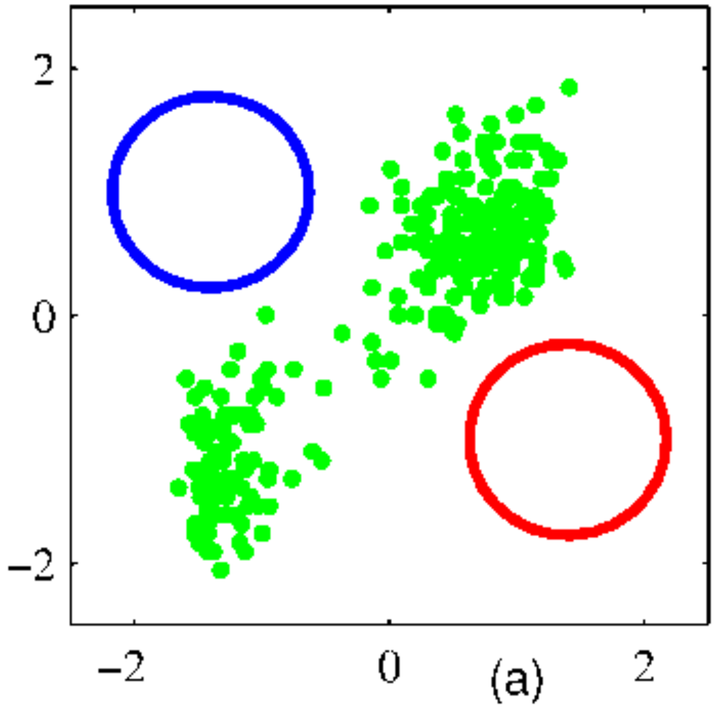
- **Supervised**
  - Classification
  - Regression
  - Data consolidation
- **Unsupervised**
  - **Clustering**
  - Dimensionality Reduction
- **Weakly supervised/semi-supervised**

  Some data supervised, some unsupervised
- **Reinforcement learning**

  Supervision: sparse reward for a sequence of decisions

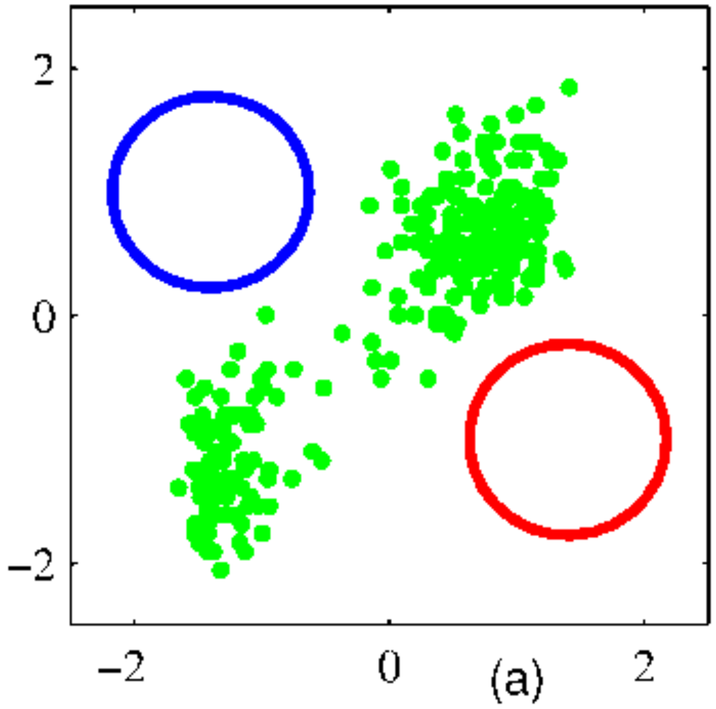
Clustering: Group Points According to X

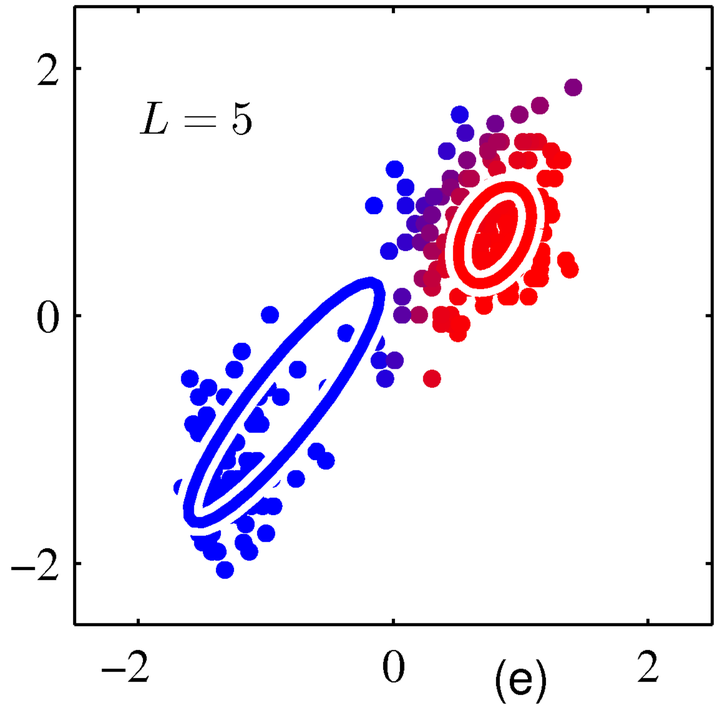
- Break a set of data into coherent groups
  - Labels are `invented’



Clustering: Group Points According to X

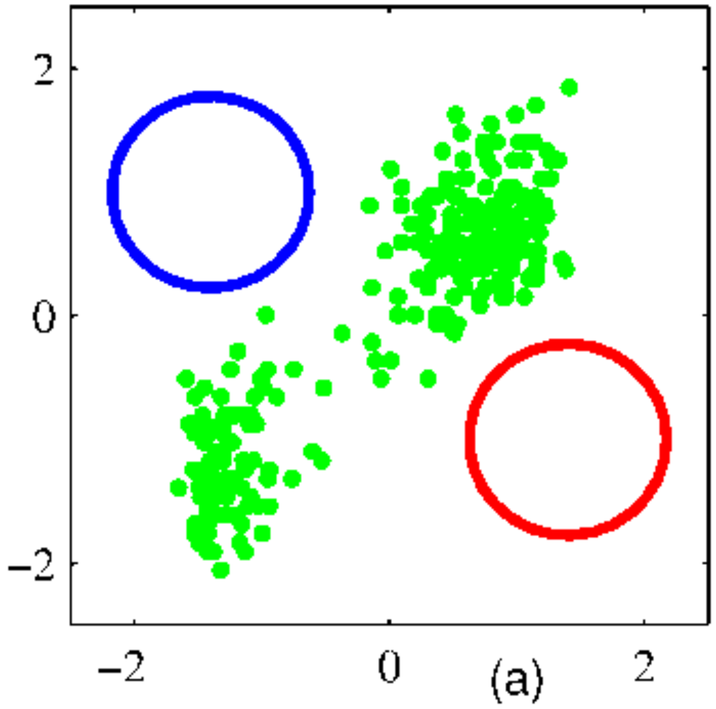
- Break a set of data into coherent groups
  - Labels are `invented'

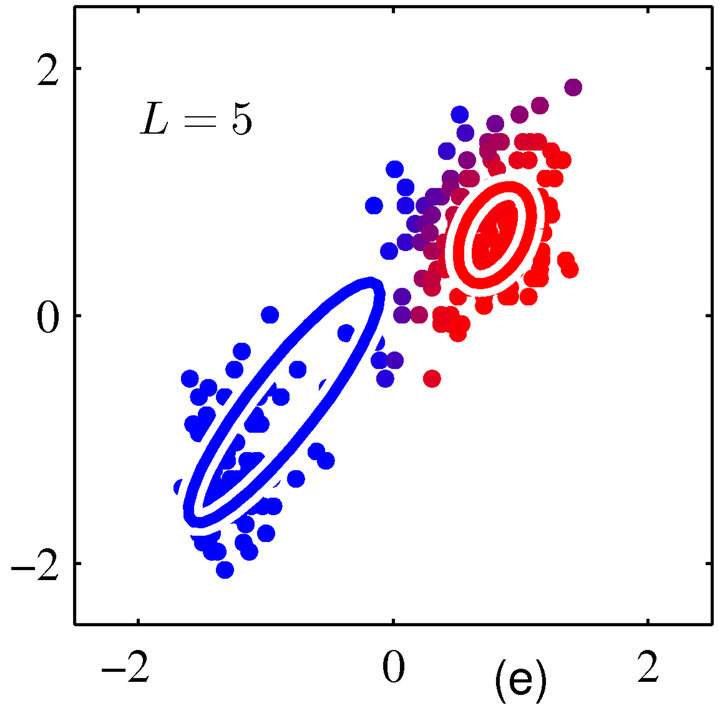


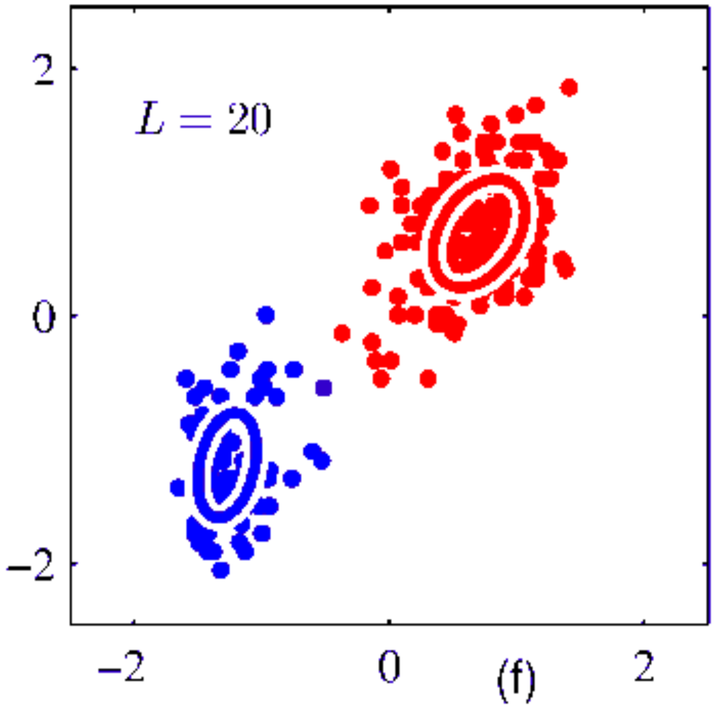


Clustering: Group Points According to X

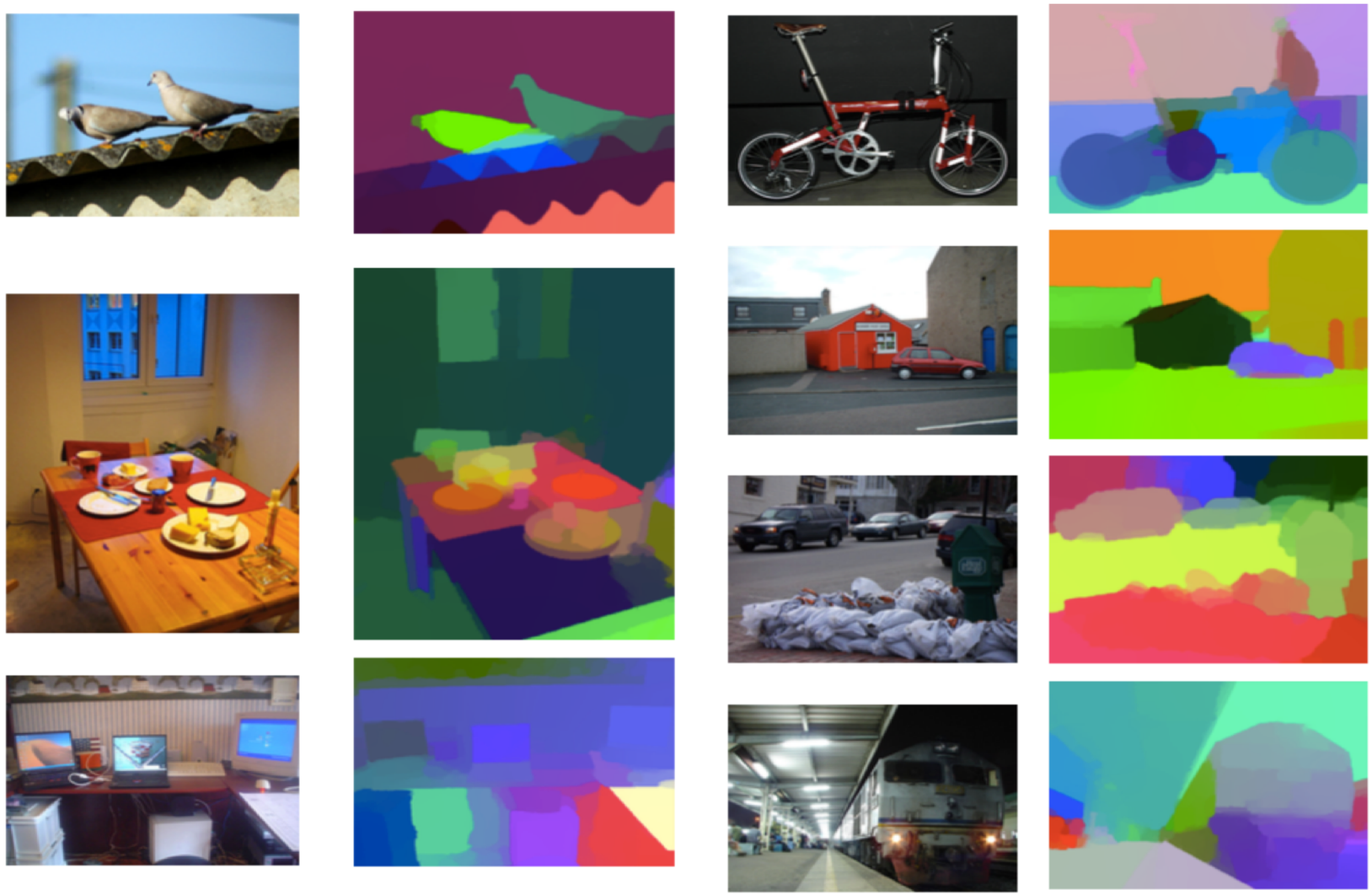
- Break a set of data into coherent groups
  - Labels are `invented'







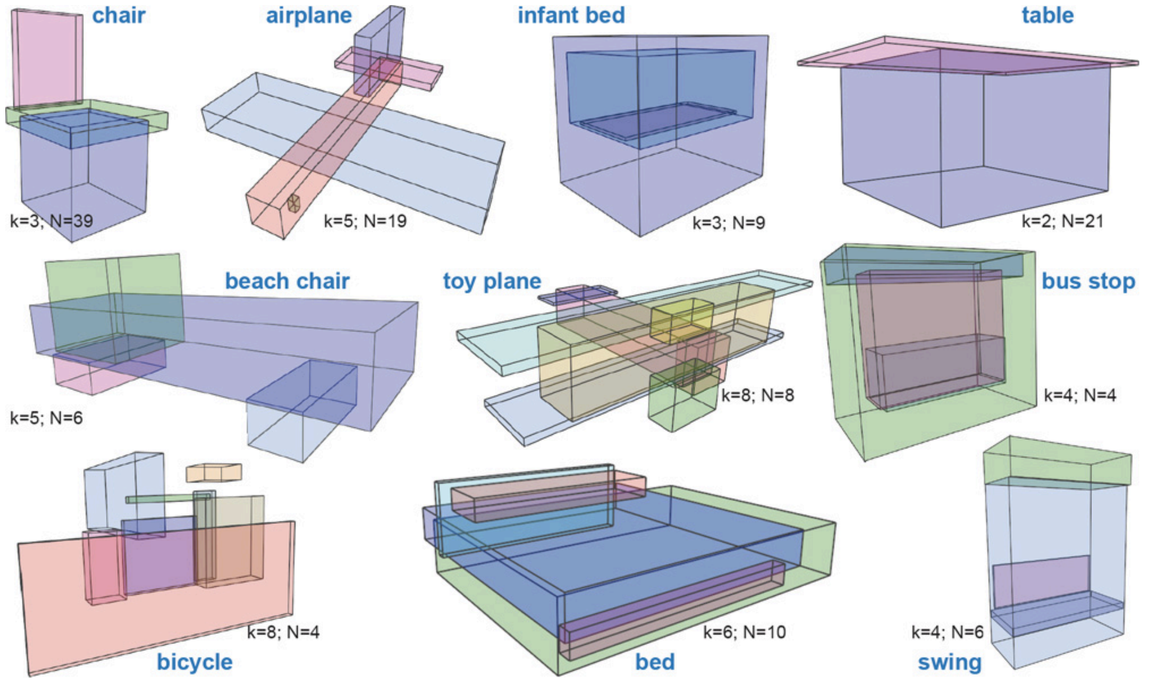
Clustering Examples: Image Segmentation using NCuts



Clustering Examples

- Spotify recommendations

[Chu et al., TVCG, 2009]



[Zheng et al., Eurographics, 2014]

Machine Learning Variants

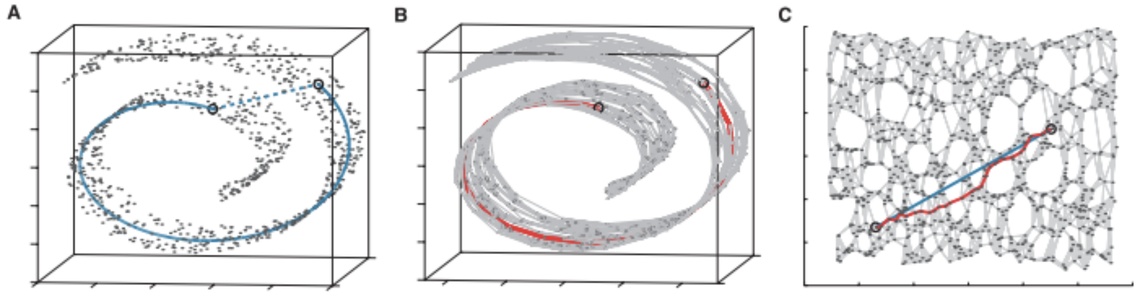
- **Supervised**
  - Classification
  - Regression
  - Data consolidation
- **Unsupervised**
  - Clustering
  - **Dimensionality Reduction**
- **Weakly supervised/semi-supervised**

  Some data supervised, some unsupervised
- **Reinforcement learning**

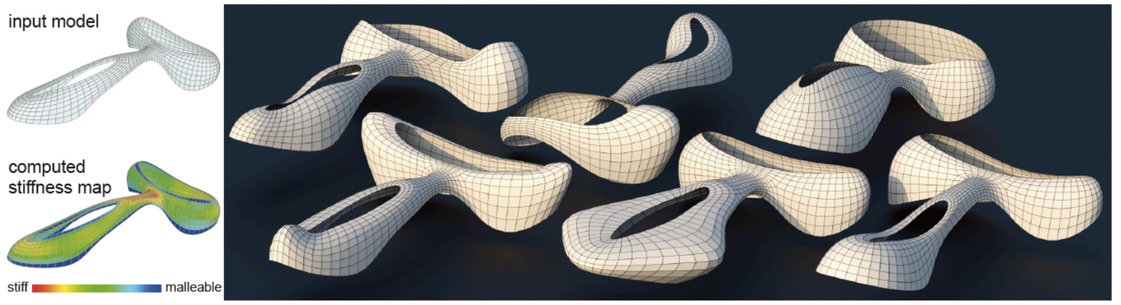
  Supervision: sparse reward for a sequence of decisions

Dimensionality Reduction (Manifold Learning)

Isomap

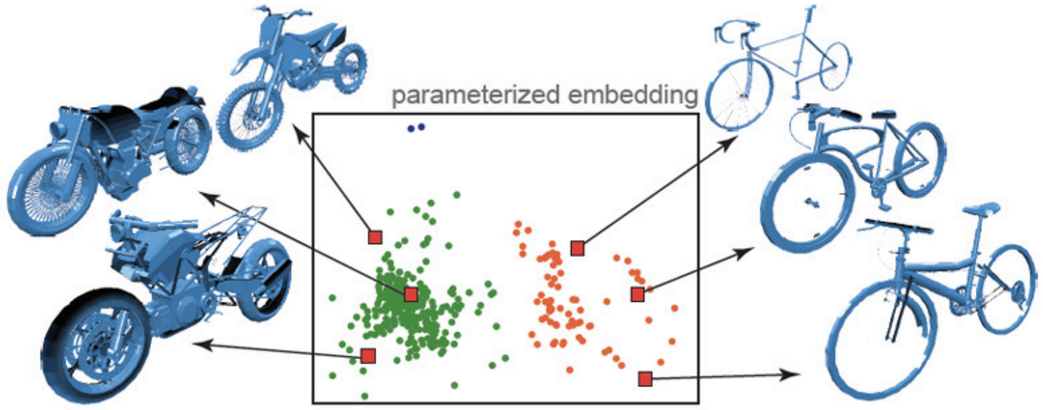


[Tenenbaum et al., Science, 2000]



[Yang et al., TOG, 2011]

Face Manifold



[Averkiou et al., Eurographics, 2014]

Example of Nonlinear Manifold: Faces

$\mathbf{x}_1$

$\mathbf{x}_2$

Example of Nonlinear Manifold: Faces

$\mathbf{x}_1$

$\frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_2)$

$\mathbf{x}_2$

Example of Nonlinear Manifold: Faces

$\mathbf{x}_1$

X

$\frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_2)$

$\mathbf{x}_2$

Moving Along Learned Face Manifold

Trajectory along the “male” dimension

[Lample et. al. Fader Networks, NIPS 2017]

Moving Along Learned Face Manifold

Trajectory along the “male” dimension

Trajectory along the “young” dimension

[Lample et. al. Fader Networks, NIPS 2017]

PCA Basis

- All eigenvalues of symmetric matrices are real.
- Any real symmetric nxn matrix has a set of **n** mutually orthogonal eigenvectors.

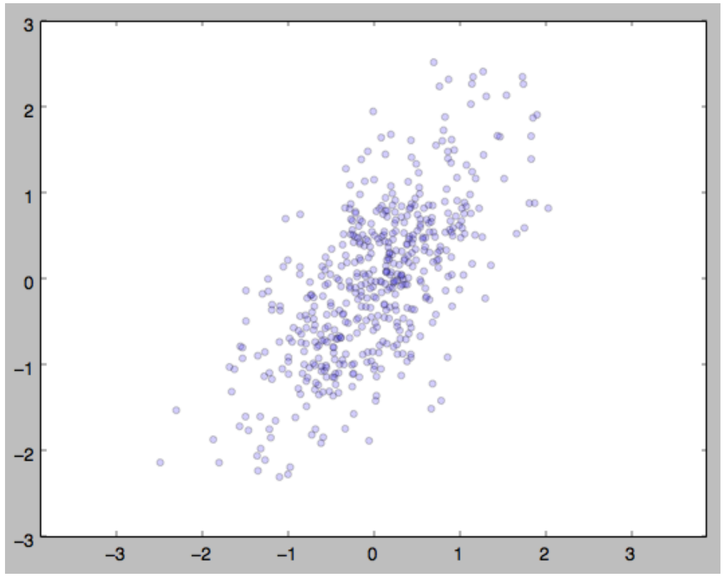
$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

$$\mathbf{A}\mathbf{e}_i = \lambda_i \mathbf{e}_i$$

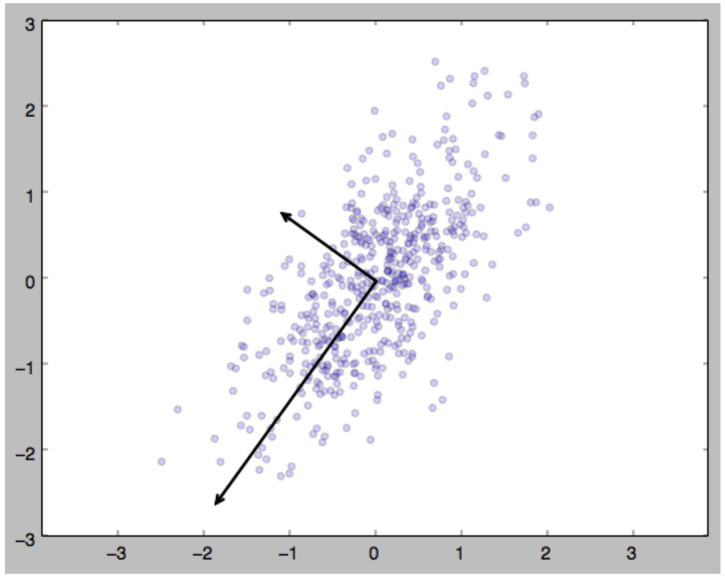
$$\mathbf{T} = [\mathbf{v}_1\ \mathbf{v}_2 \dots]$$

$$\mathbf{T}^{-1} A \mathbf{T} = \mathrm{diag}(\lambda_1, \lambda_2, \dots)$$

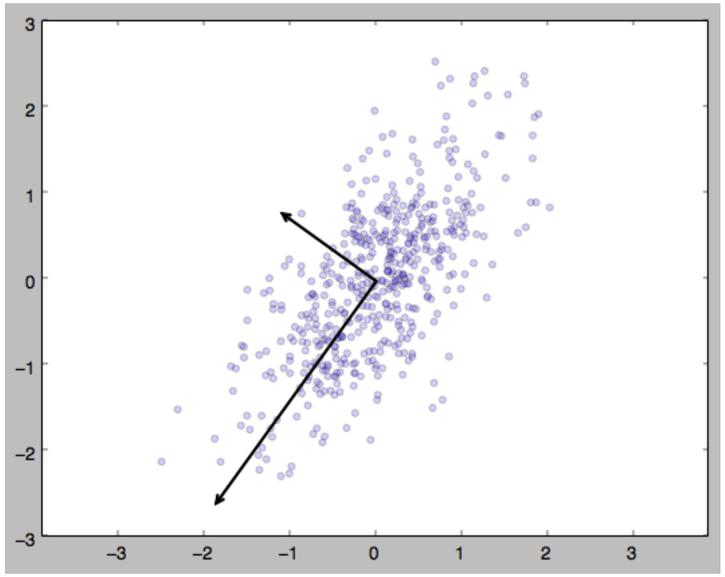
Code Example



Code Example

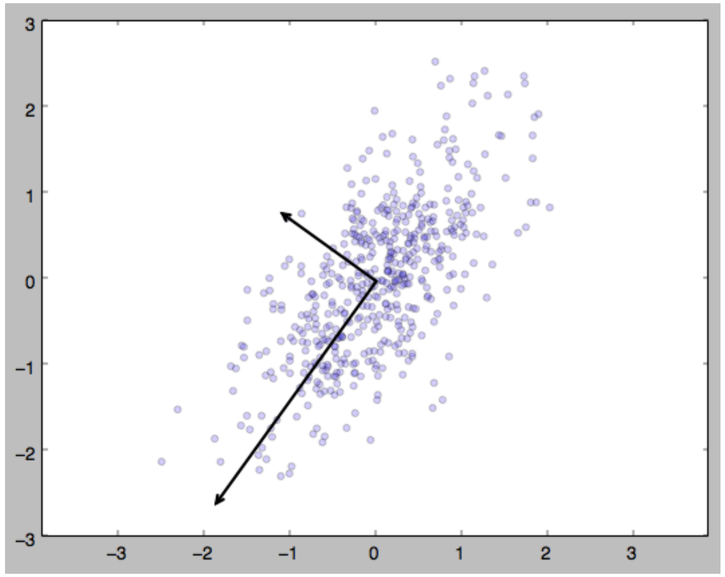


Code Example



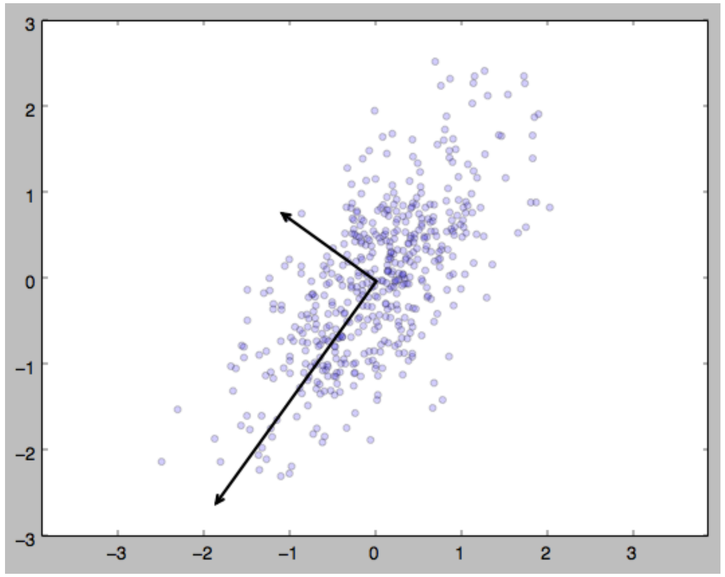

```
rng = np.random.RandomState(10)
X = np.dot(rng.rand(2, 2), rng.randn(2, 500)).T

mean_vec = np.mean(X, axis=0)
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)
```

Code Example



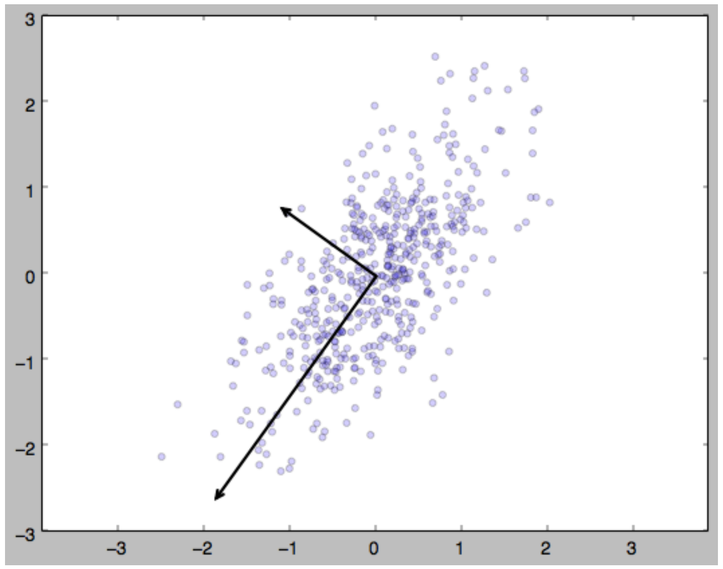

```
rng = np.random.RandomState(10)
X = np.dot(rng.rand(2, 2), rng.randn(2, 500)).T

mean_vec = np.mean(X, axis=0)
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)
```

Code Example



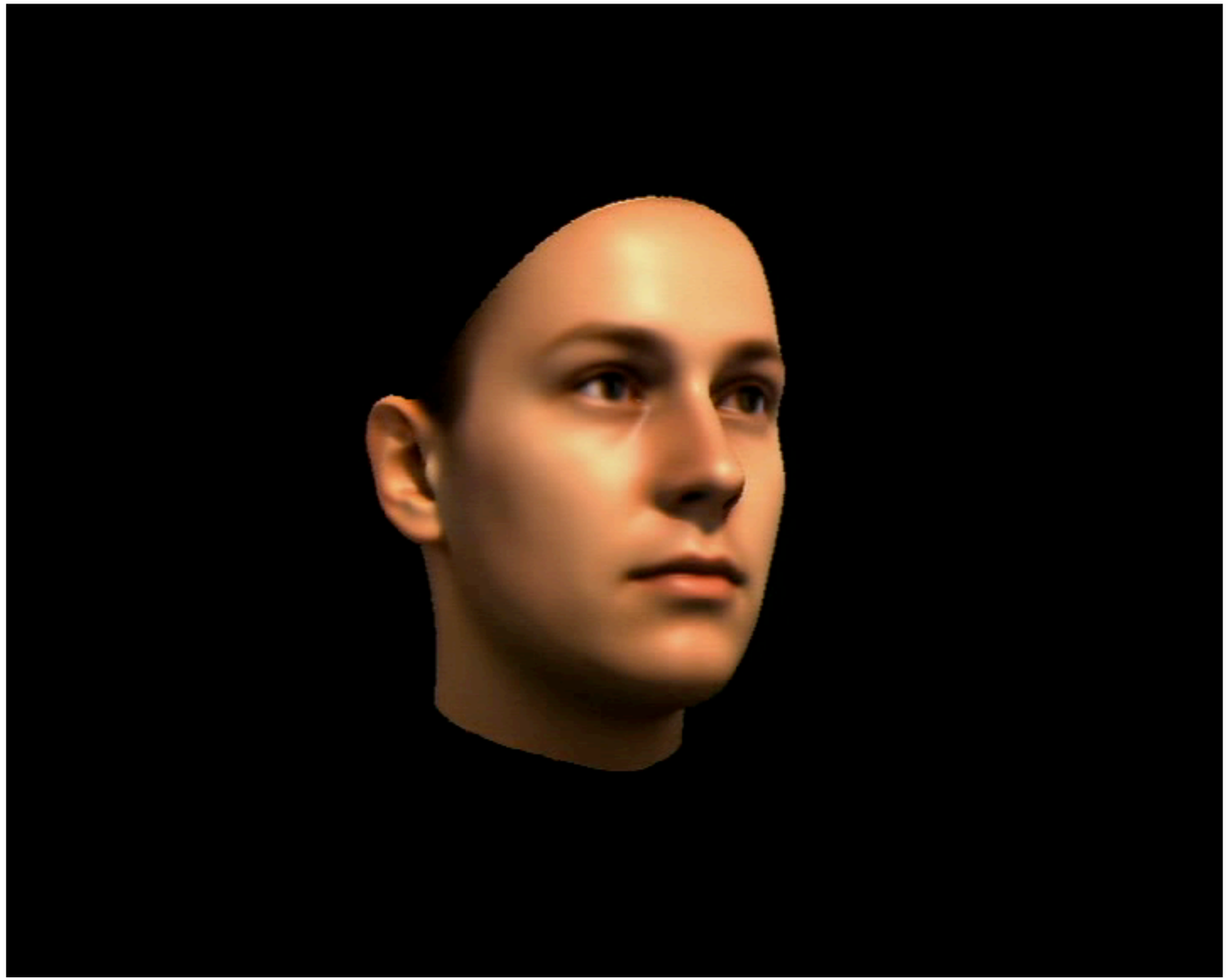

```
rng = np.random.RandomState(10)
X = np.dot(rng.rand(2, 2), rng.randn(2, 500)).T

mean_vec = np.mean(X, axis=0)
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)
```

Code Example

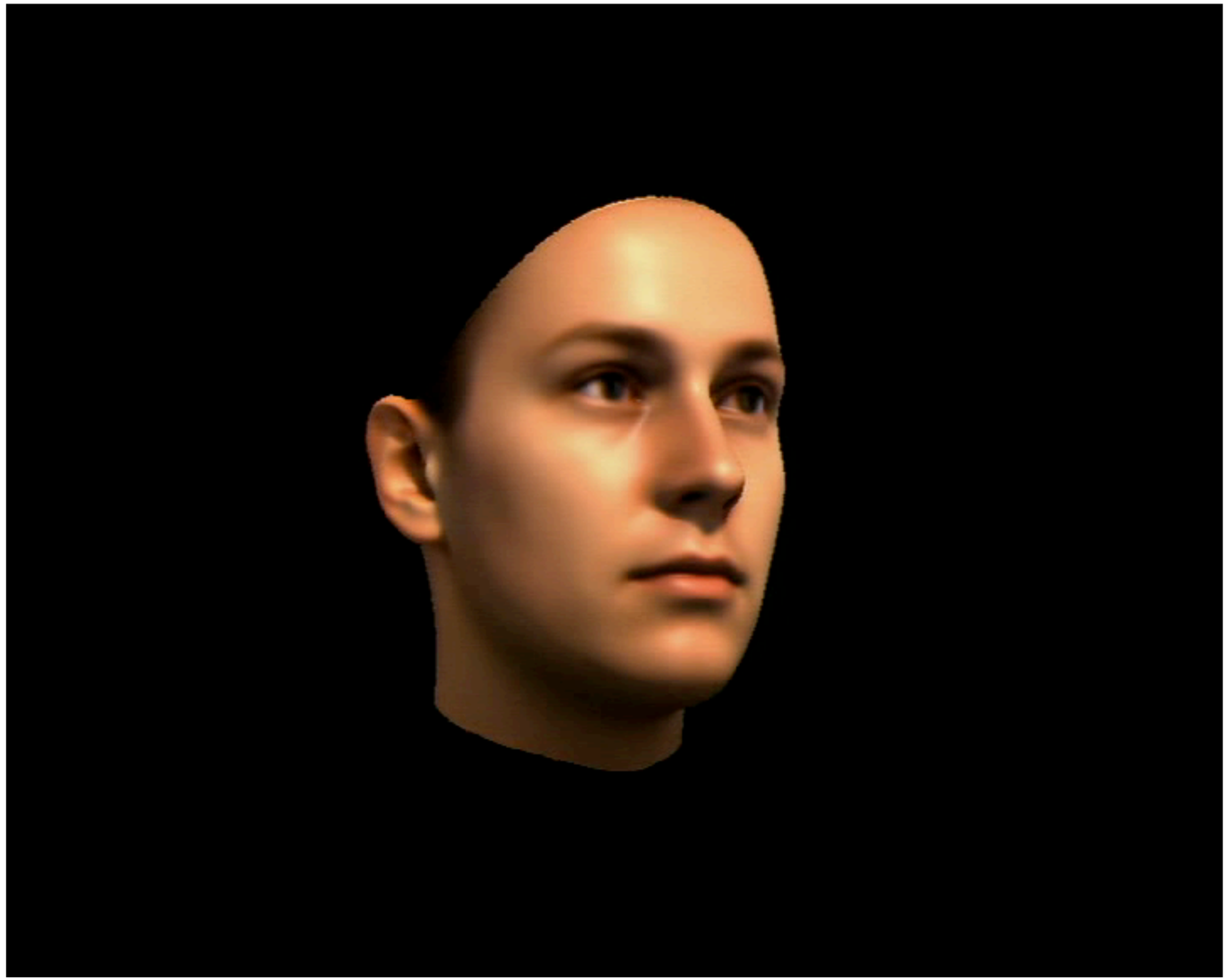



```
mean_vec = np.mean(X, axis=0)
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)
matU, sigma, matV = np.linalg.svd(cov_mat)
```

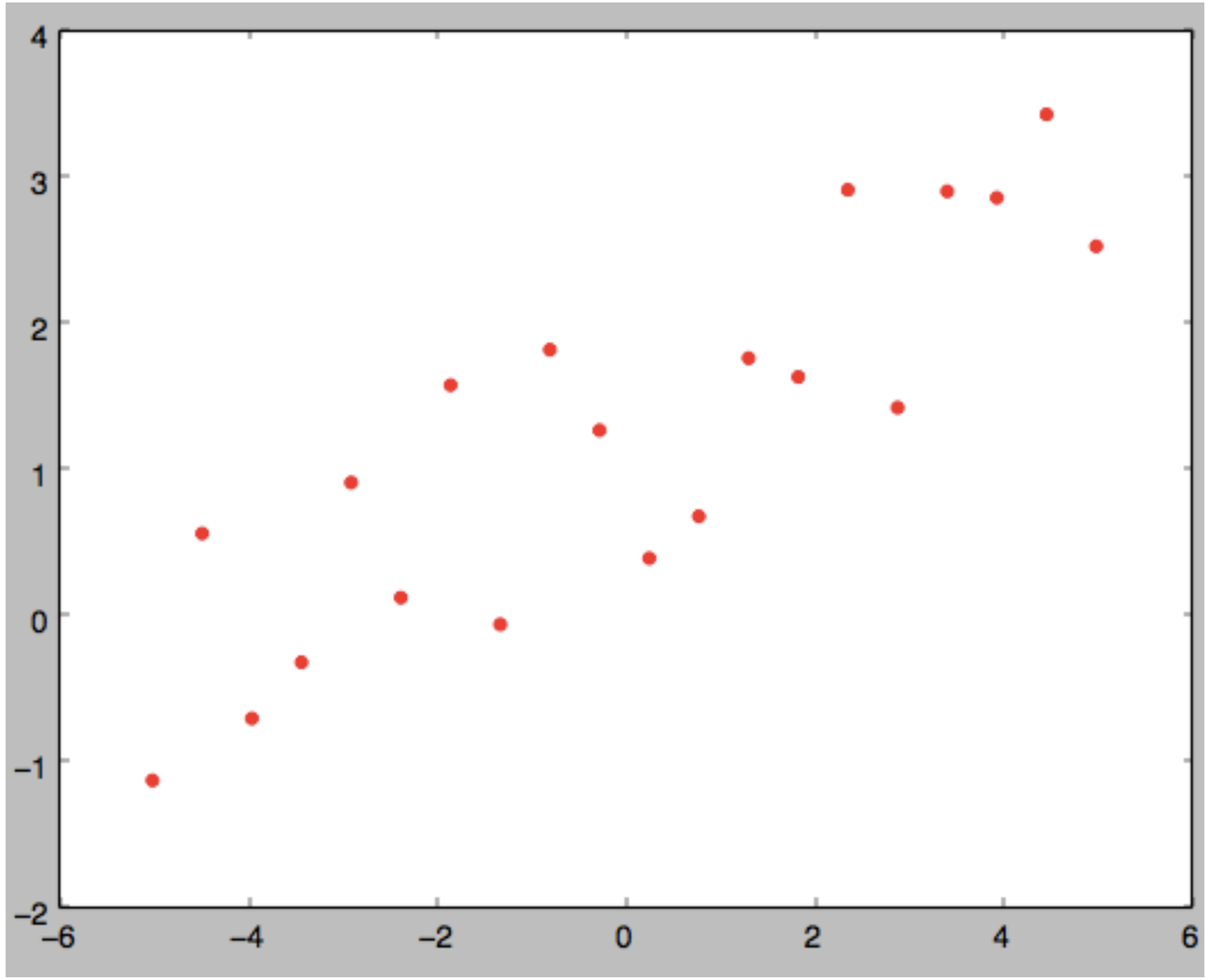
Morphable Faces



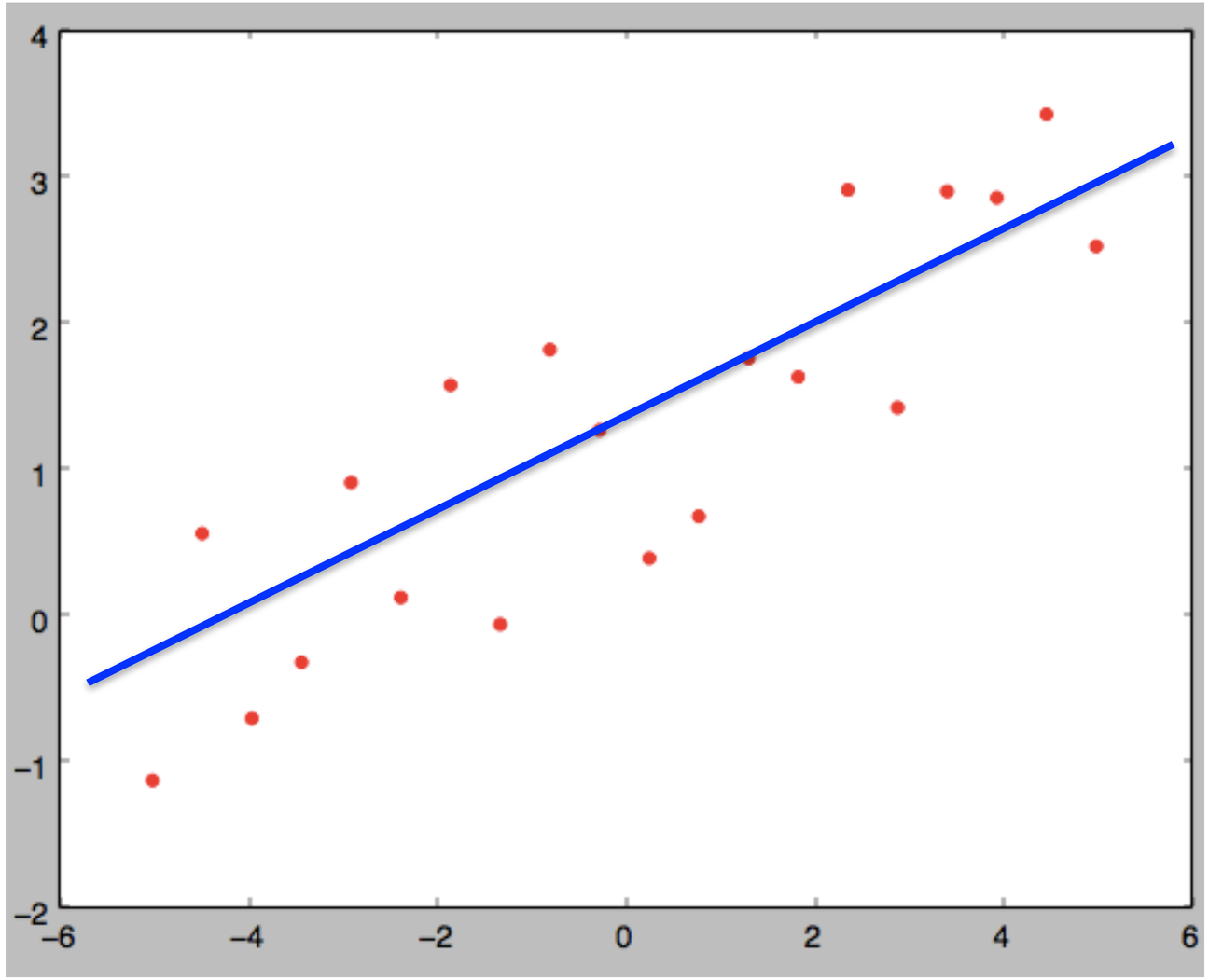
Morphable Faces



Regression: Continuous Output

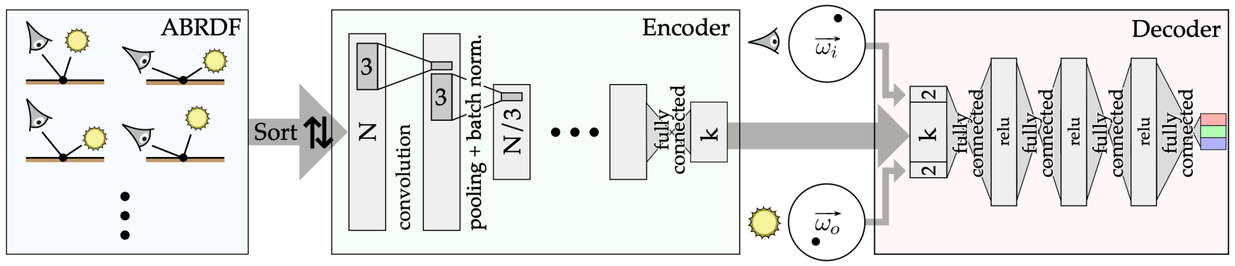


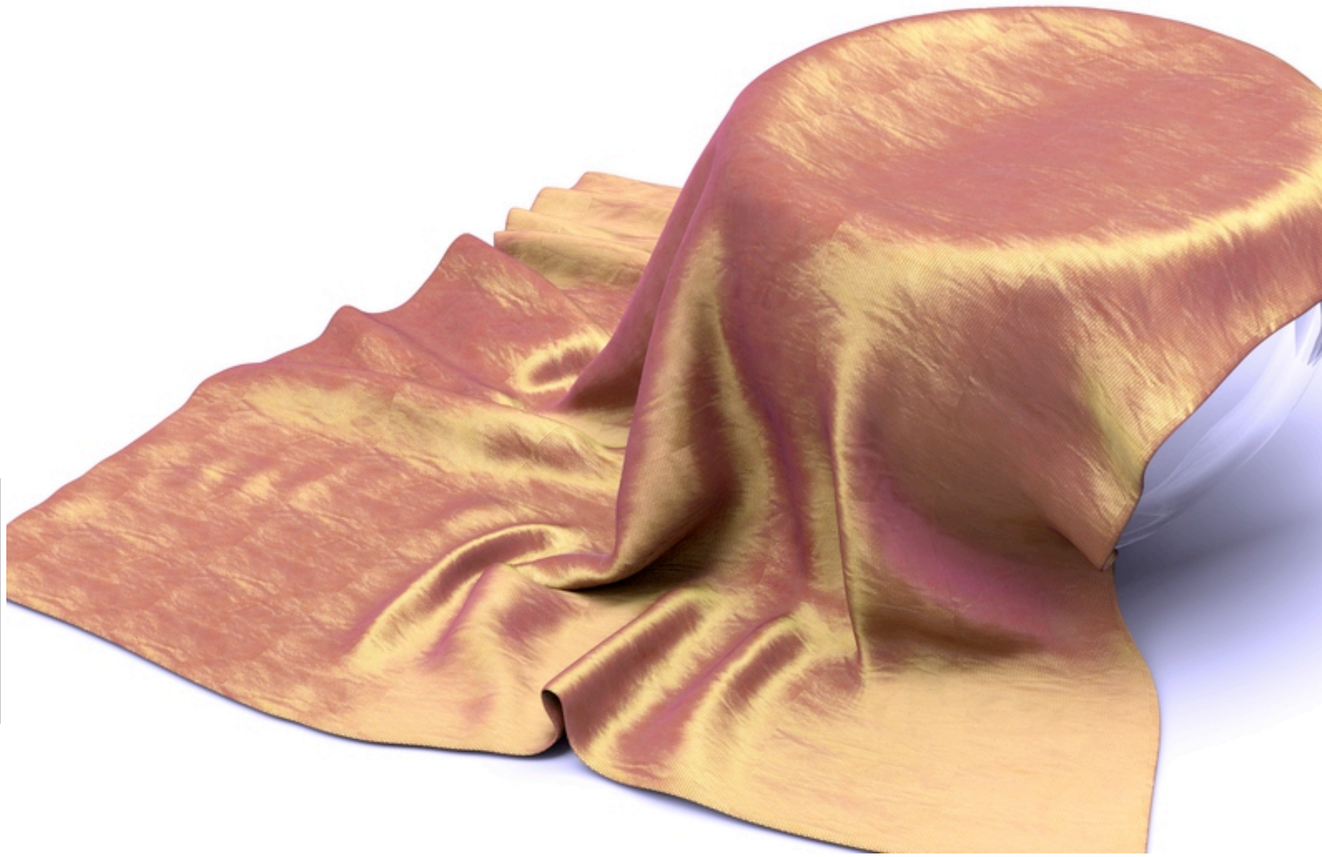
Regression: Continuous Output



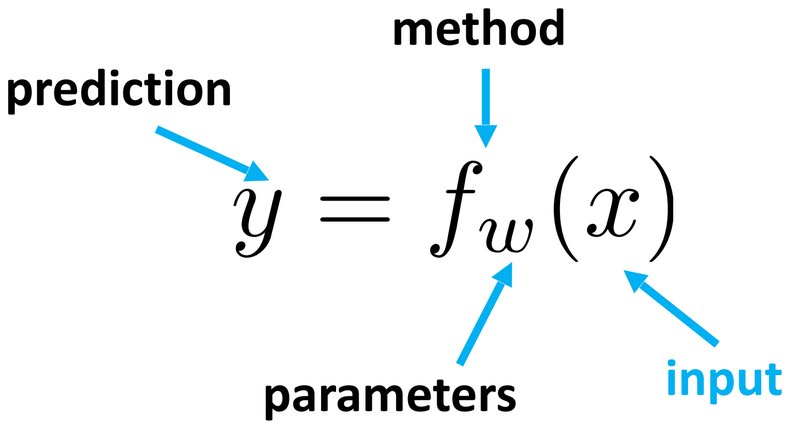
Neural BTF Compression and Interpolation

[Rainer et al. 2019, EG]





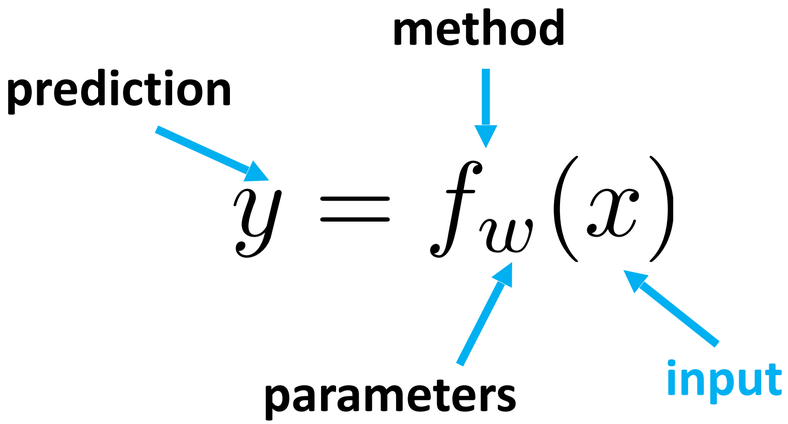
Learning a Function



Calculus $x \in \mathbb{R}$

Vector calculus $\mathbf{x} \in \mathbb{R}^d$

Learning a Function

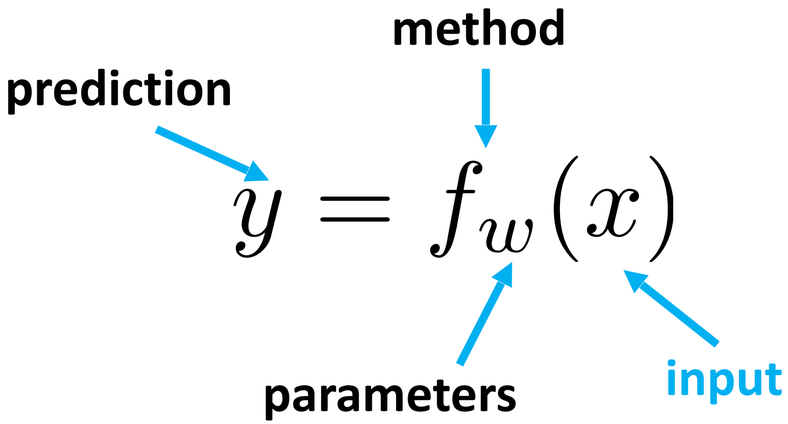


Calculus $x \in \mathbb{R}$

Vector calculus $\mathbf{x} \in \mathbb{R}^d$

Machine learning: can work also for discrete inputs, strings, images, meshes, animations, …

Learning a Function



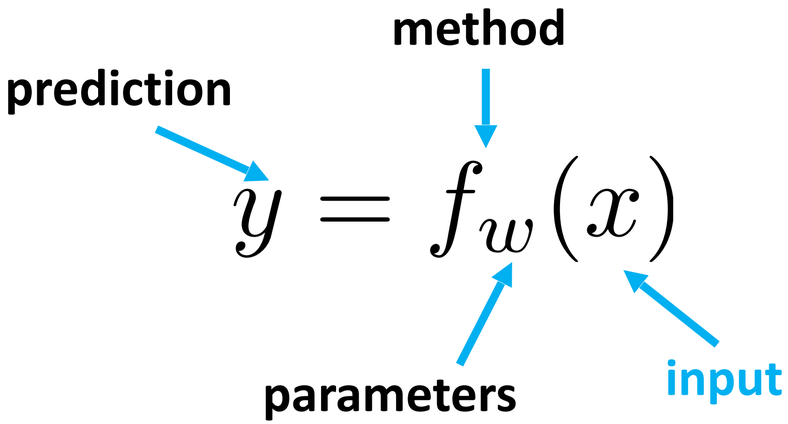
Calculus $x \in \mathbb{R}$

Vector calculus $\mathbf{x} \in \mathbb{R}^d$

Classification: $y \in \{0, 1\}$

Machine learning: can work also for discrete inputs, strings, images, meshes, animations, …

Learning a Function



Calculus $x \in \mathbb{R}$

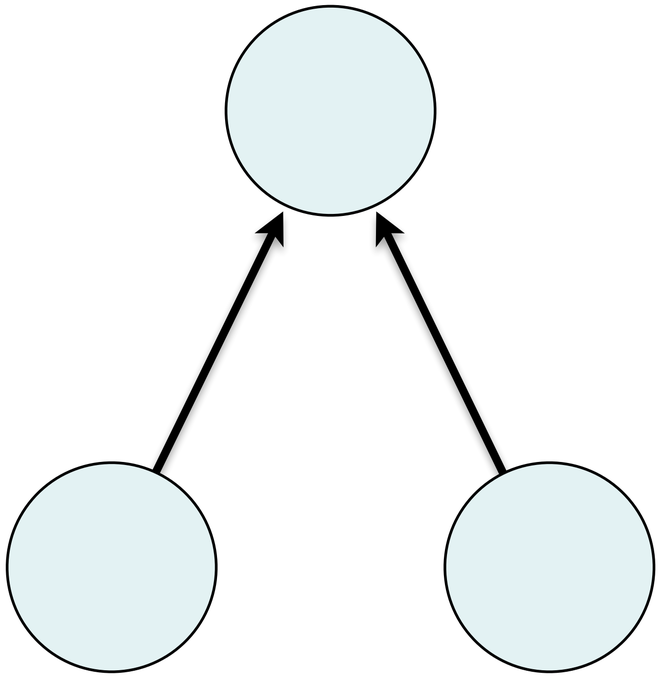
Vector calculus $\mathbf{x} \in \mathbb{R}^d$

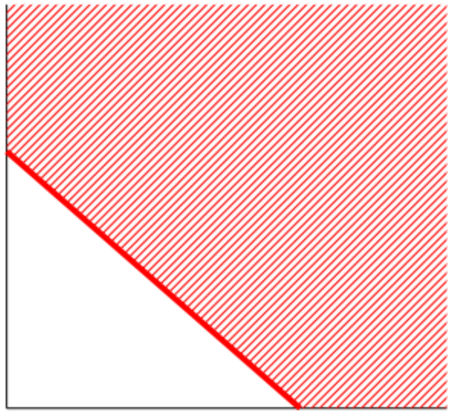
Classification: $y \in \{0, 1\}$

Regression: $y \in \mathbb{R}$

Machine learning: can work also for discrete inputs, strings, images, meshes, animations, …

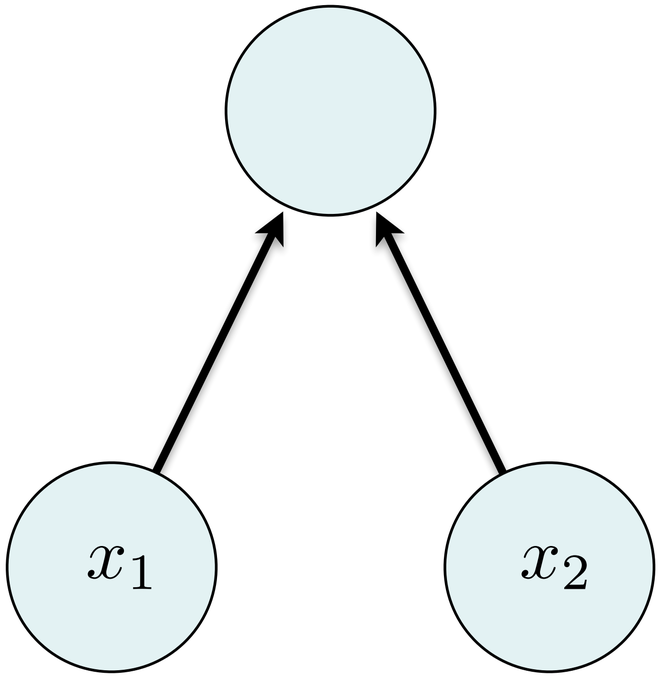
Learning a Linear Separator/Classifier

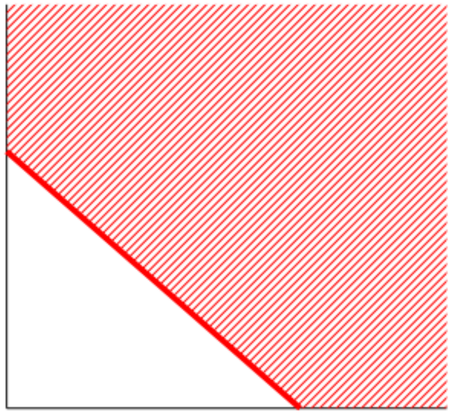




separating hyperplane

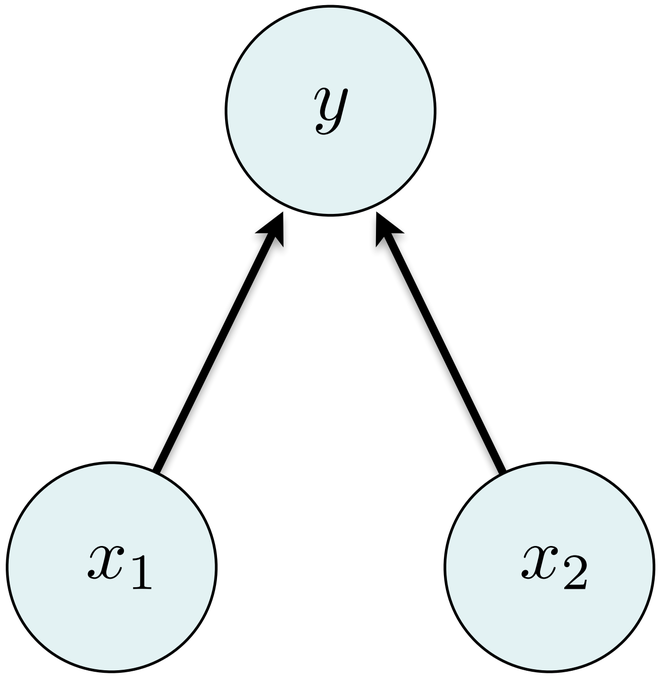
Learning a Linear Separator/Classifier

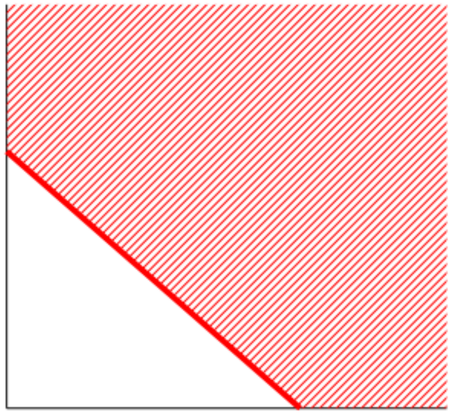




separating hyperplane

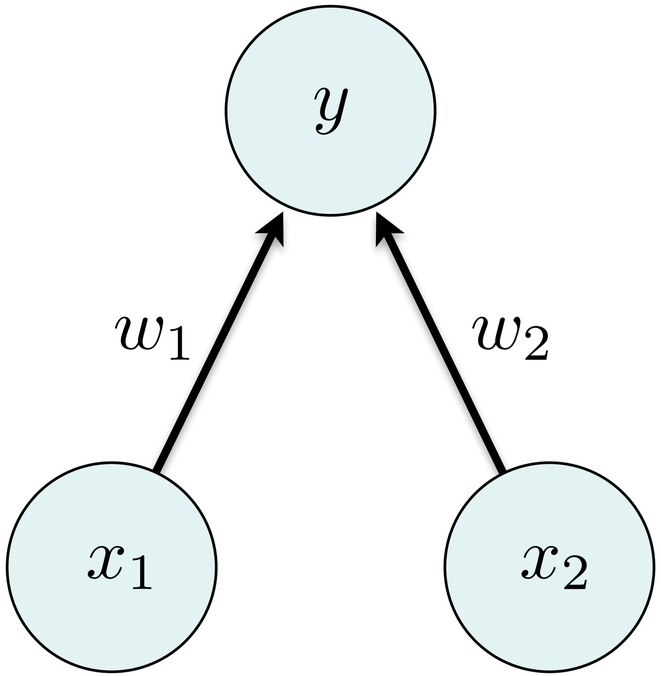
Learning a Linear Separator/Classifier

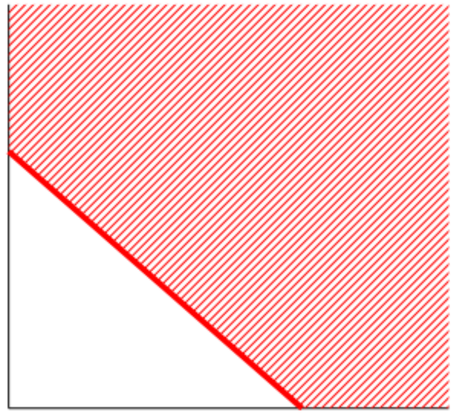




separating hyperplane

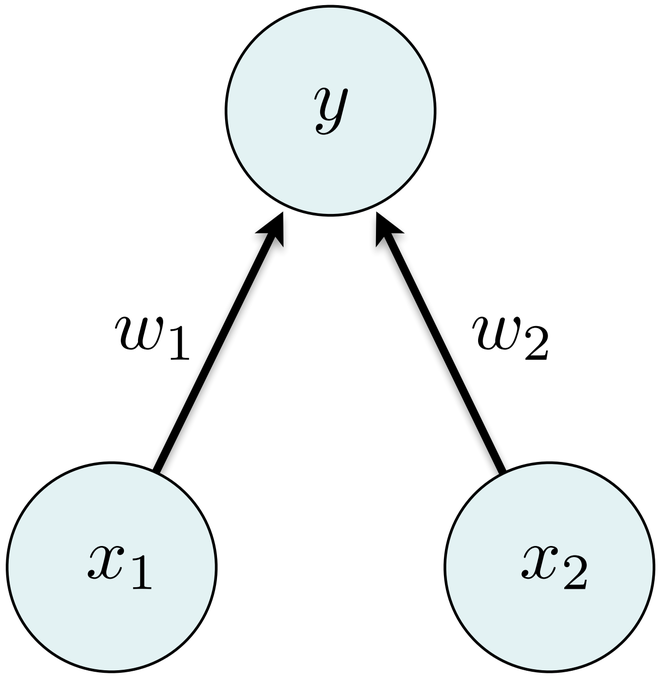
Learning a Linear Separator/Classifier

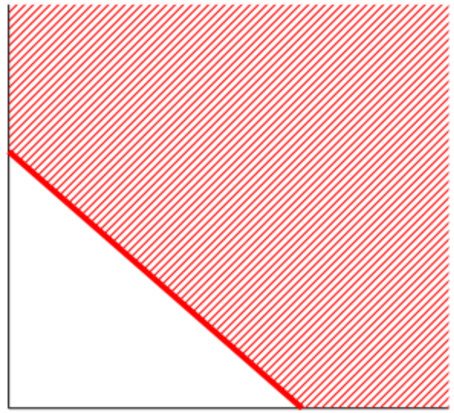




separating hyperplane

Learning a Linear Separator/Classifier

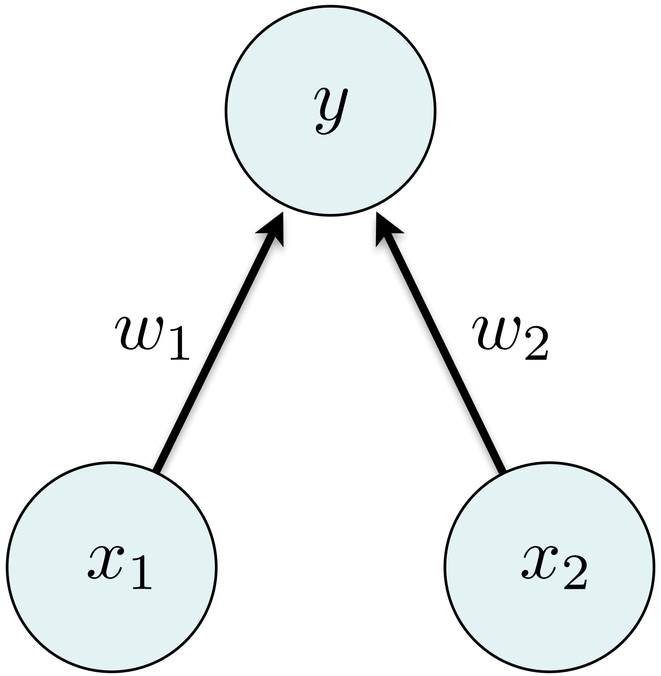


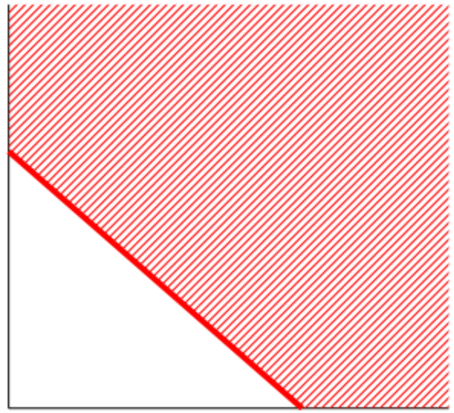


separating hyperplane

$$y = f(w_1 x_1 + w_2 x_2)$$

Learning a Linear Separator/Classifier

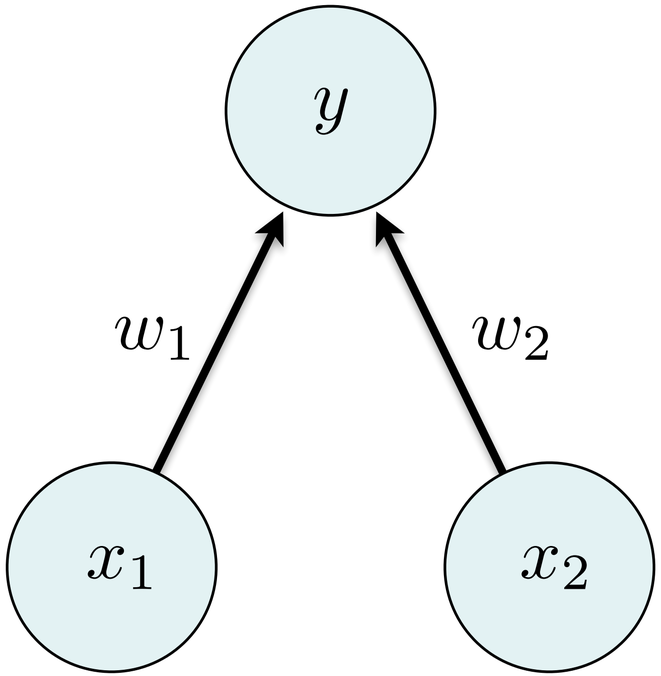


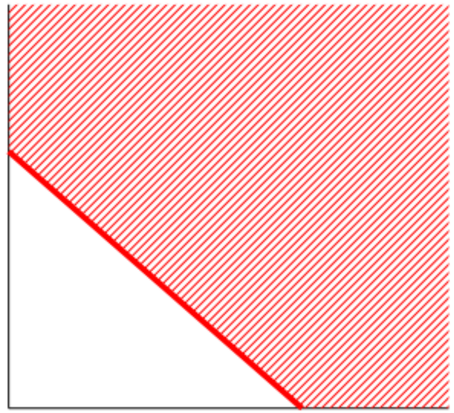


separating hyperplane

$$y = f(w_1 x_1 + w_2 x_2) = \mathcal{H}(w_1 x_1 + w_2 x_2)$$

Learning a Linear Separator/Classifier



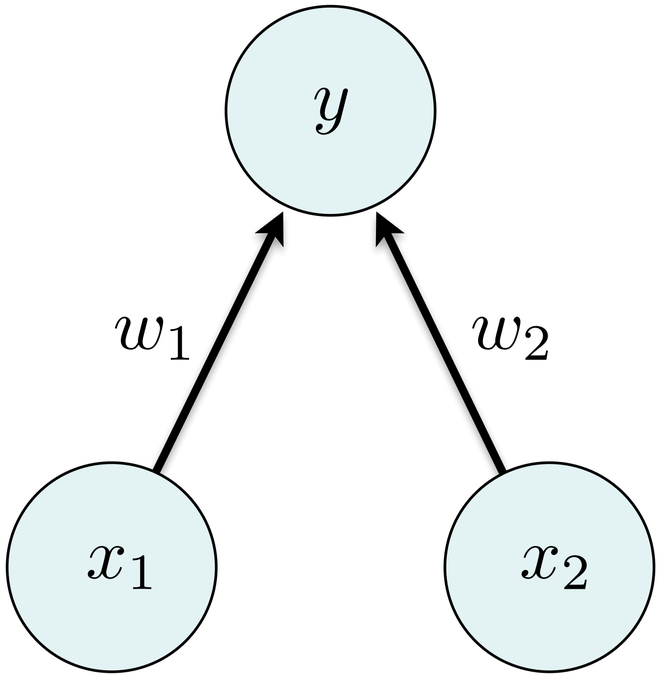


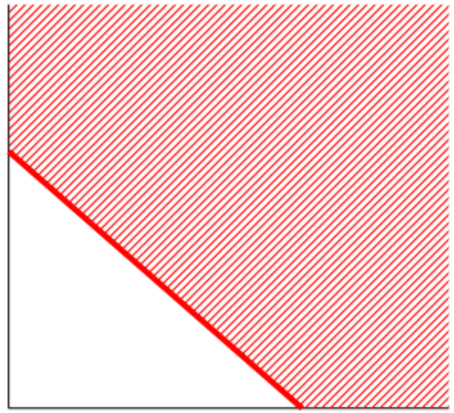
separating hyperplane

fixed non-linearity

$$y = f(w_1 x_1 + w_2 x_2) = \mathcal{H}(w_1 x_1 + w_2 x_2)$$

Learning a Linear Separator/Classifier





separating hyperplane

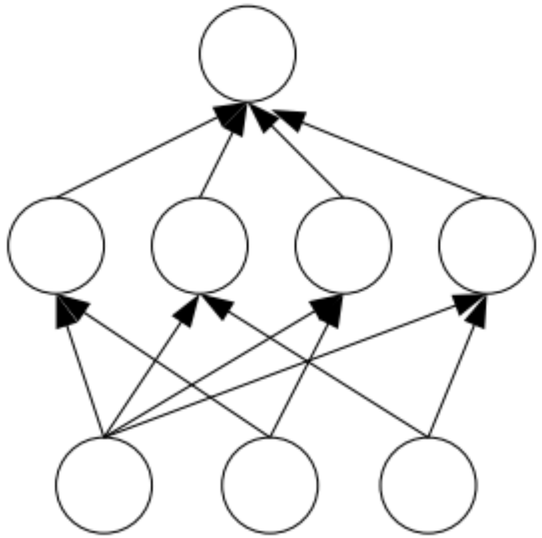
fixed non-linearity

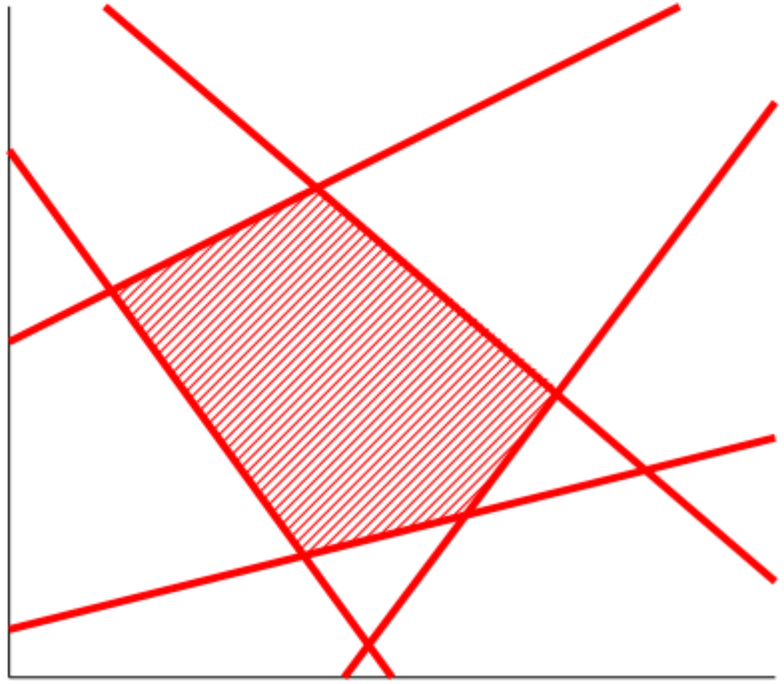
learned

$$y = f(w_1 x_1 + w_2 x_2) = \mathcal{H}(w_1 x_1 + w_2 x_2)$$

Combining Simple Functions/Classifiers

2 layers of trainable weights

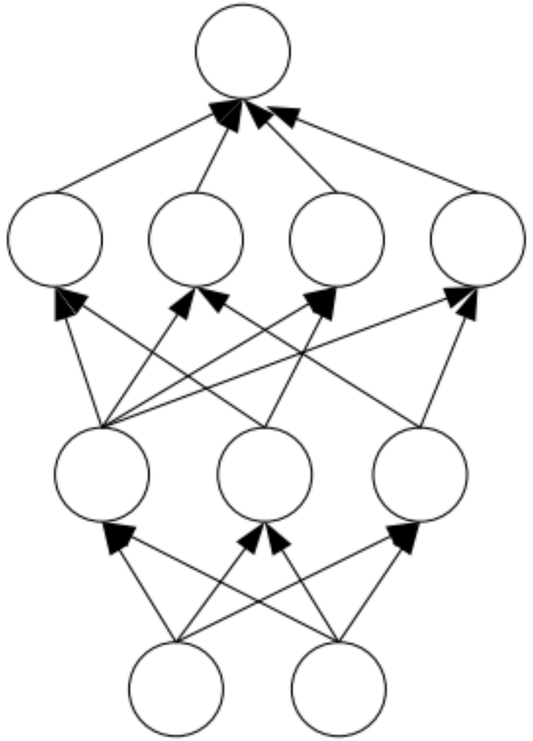


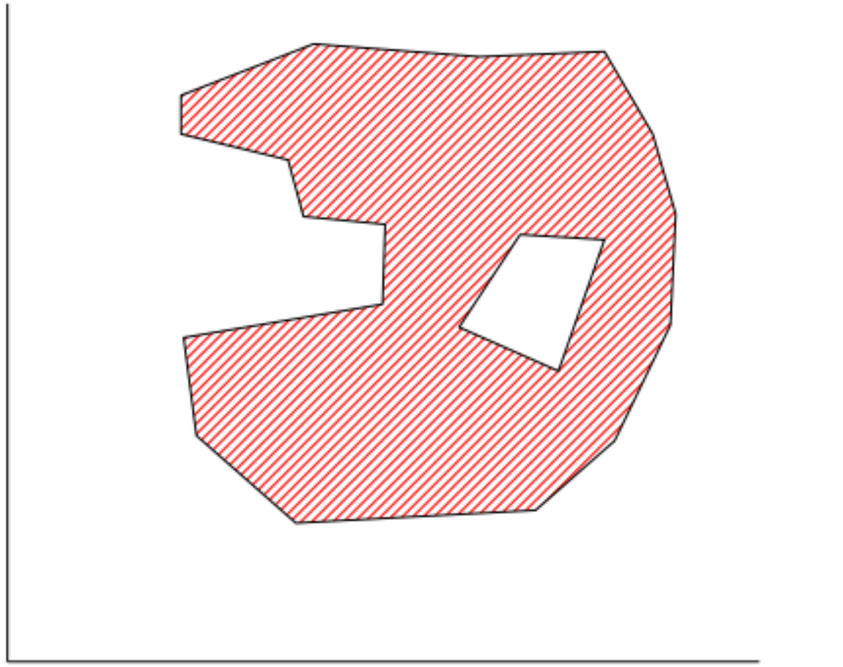


convex polygon region

Combining Simple Functions/Classifiers

3 layers of trainable weights





composition of polygons: convex regions

Regression

1. Least Squares fitting

2. Nonlinear error function and gradient descent

3. Perceptron training (simple neural network)

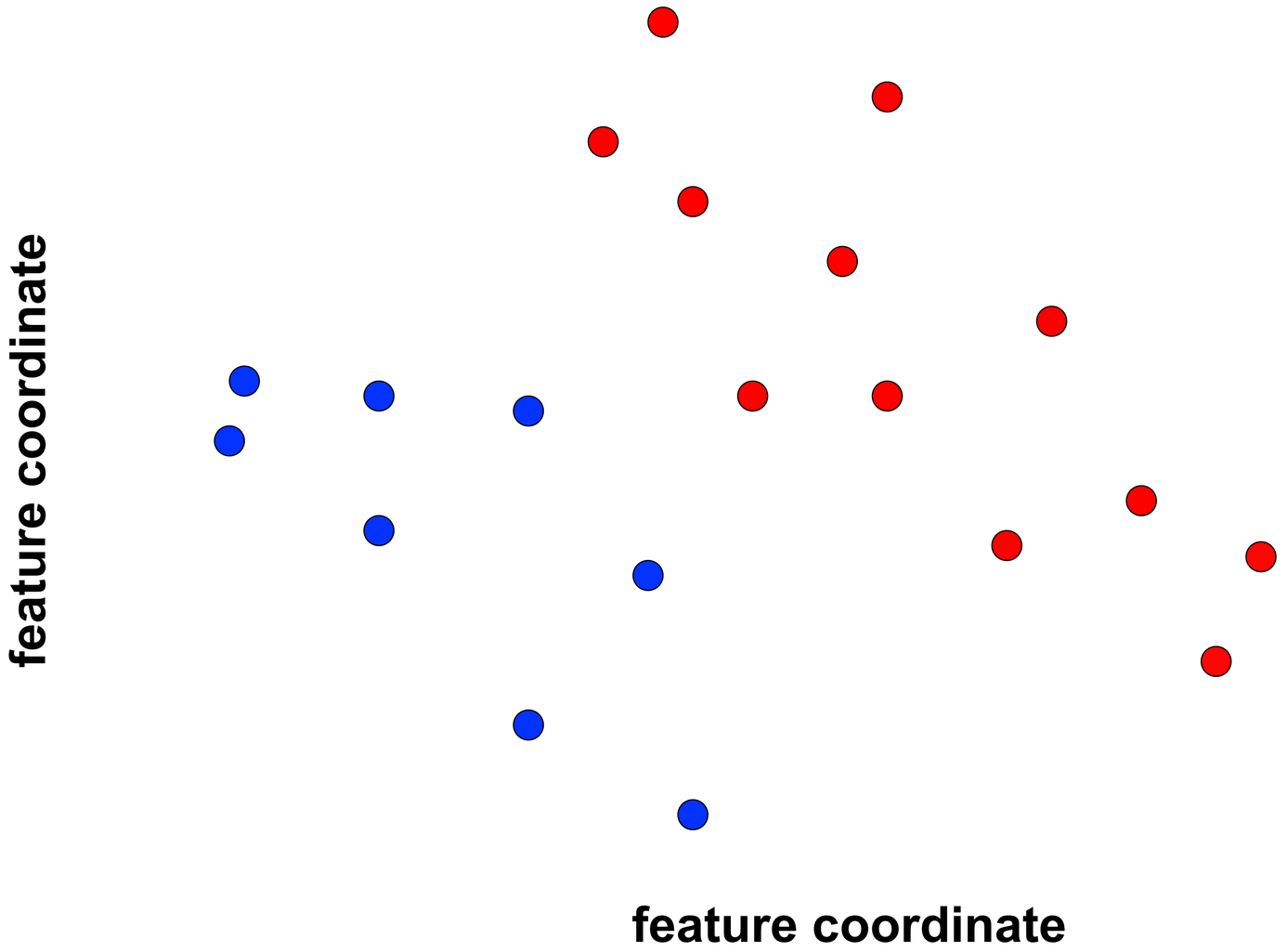
Regression

1. **Least Squares fitting**

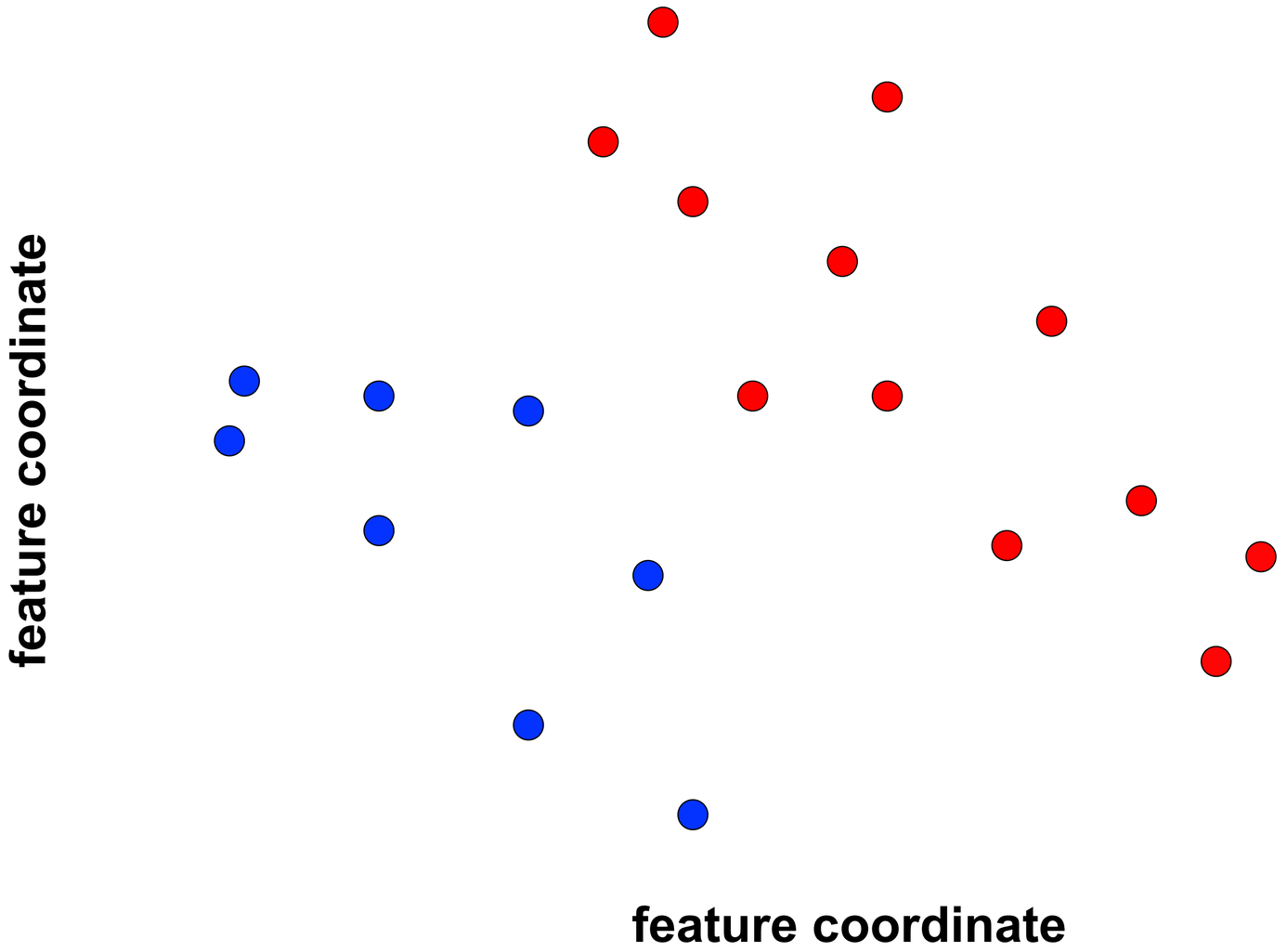
2. Nonlinear error function and gradient descent

3. Perceptron training (simple neural network)

Reminder: Linear Classifier



Reminder: Linear Classifier

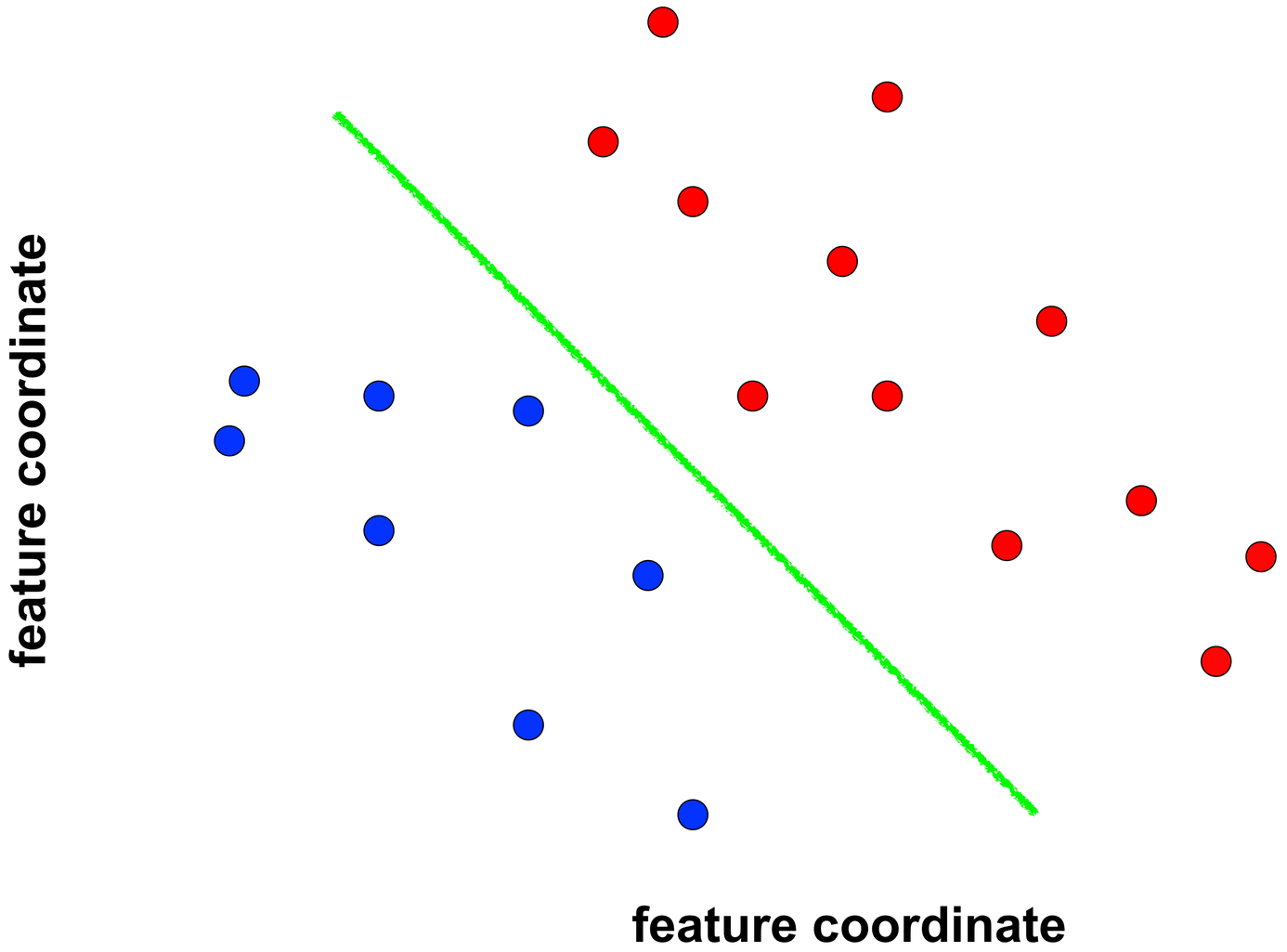


supervised setting

labelled input

$$y_t = \begin{cases} +1 & \bullet \\ -1 & \bullet \end{cases}$$

Reminder: Linear Classifier

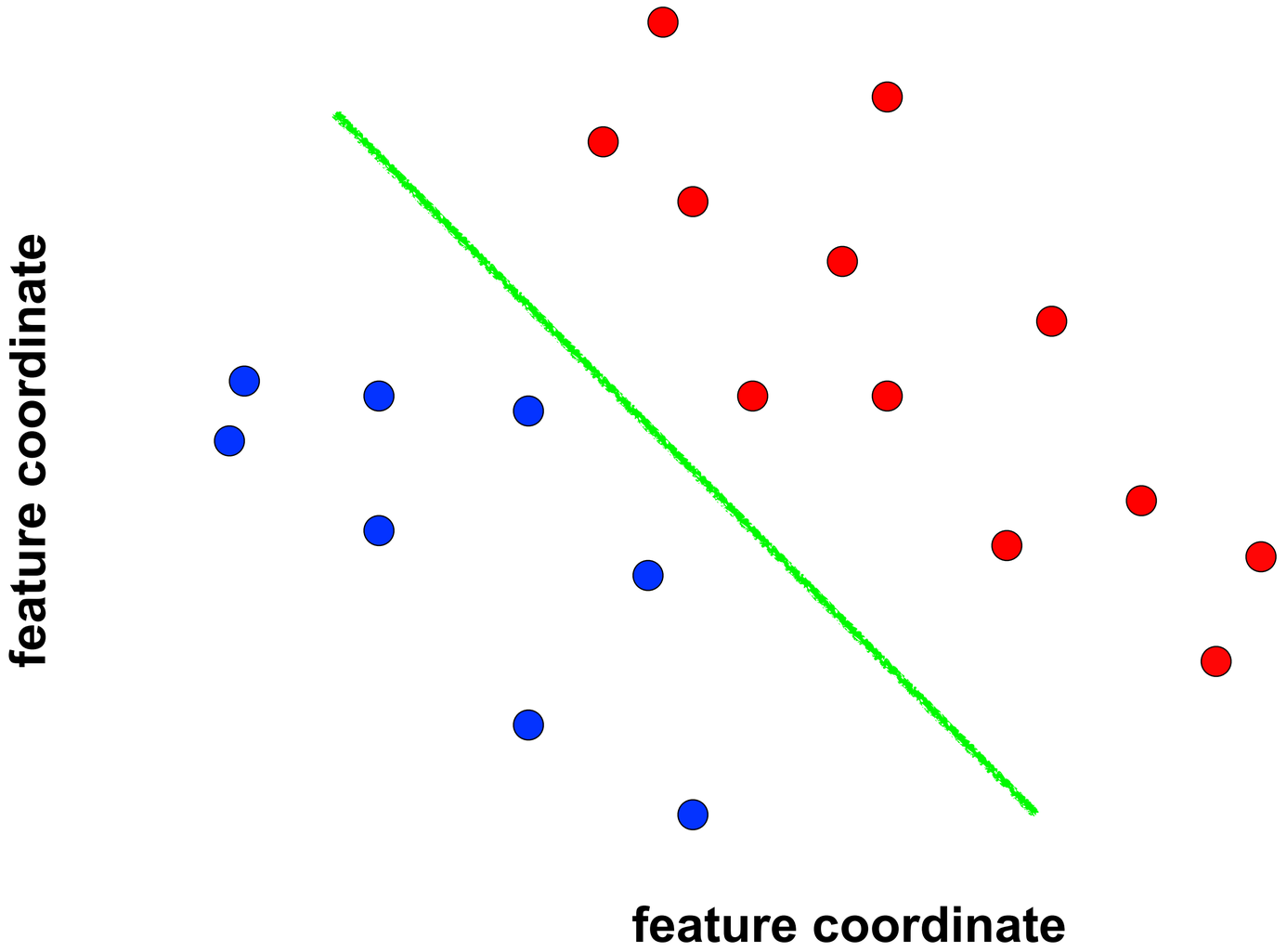


supervised setting

labelled input

$$y_t = \begin{cases} +1 & \bullet \\ -1 & \bullet \end{cases}$$

Reminder: Linear Classifier



$\mathbf{x}_i$ positive: $\mathbf{x}_i \cdot \mathbf{w} \geq 0$

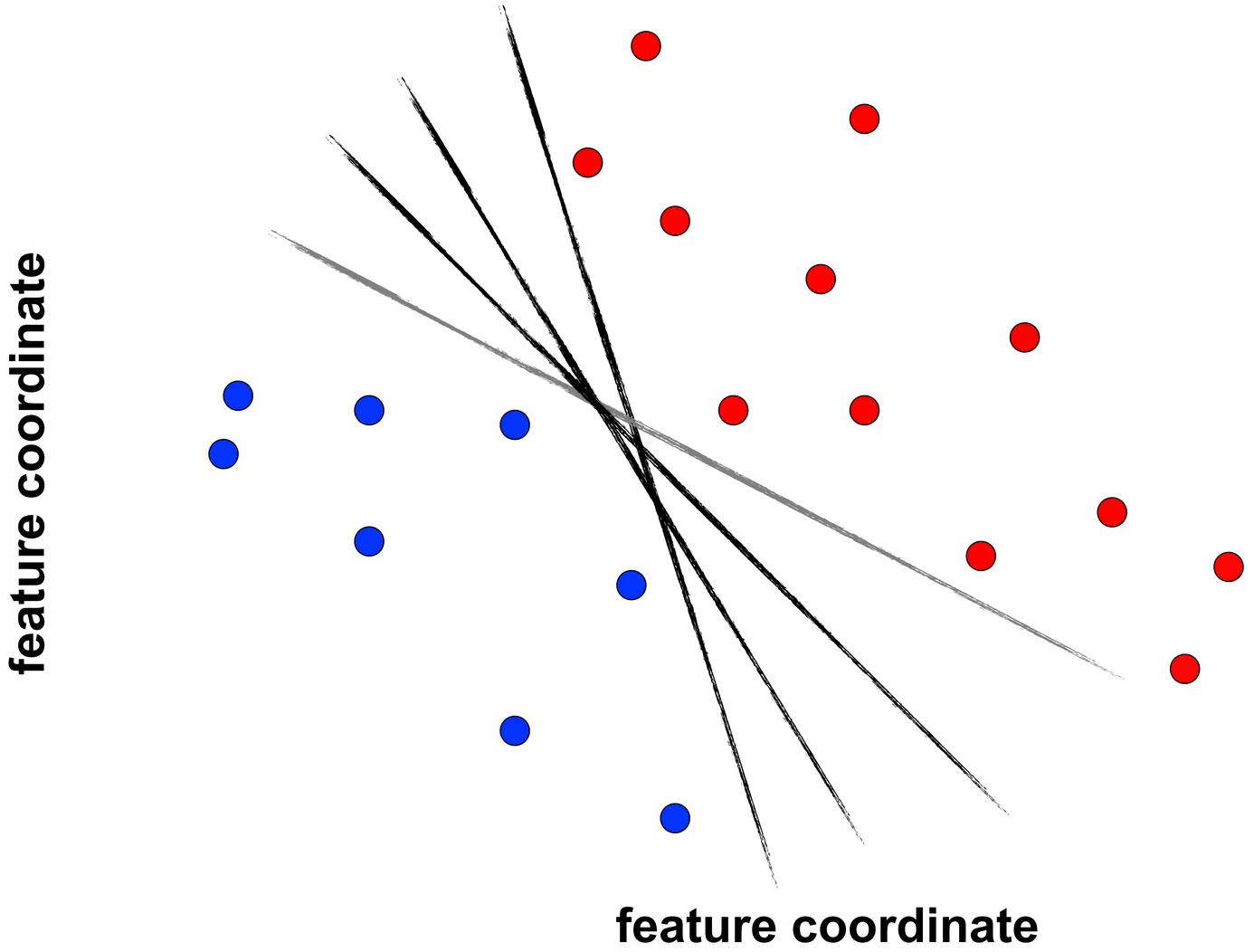
$\mathbf{x}_i$ negative: $\mathbf{x}_i \cdot \mathbf{w} < 0$

supervised setting

labelled input

$$y_t = \begin{cases} +1 & \bullet \\ -1 & \bullet \end{cases}$$

Which Line to Pick?



$\mathbf{x}_i$ positive: $\mathbf{x}_i \cdot \mathbf{w} \geq 0$

$\mathbf{x}_i$ negative: $\mathbf{x}_i \cdot \mathbf{w} < 0$

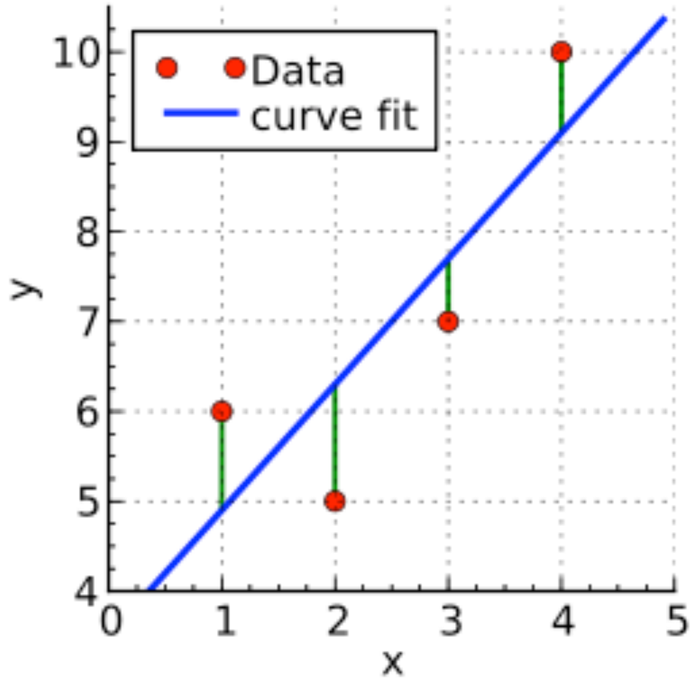
supervised setting

labelled input

$$y_t = \begin{cases} +1 & \bullet \\ -1 & \bullet \end{cases}$$

Sum of Square Errors *(MSE without the mean)*

$$y^i = \mathbf{w}^T \mathbf{x}^i + \epsilon^i$$

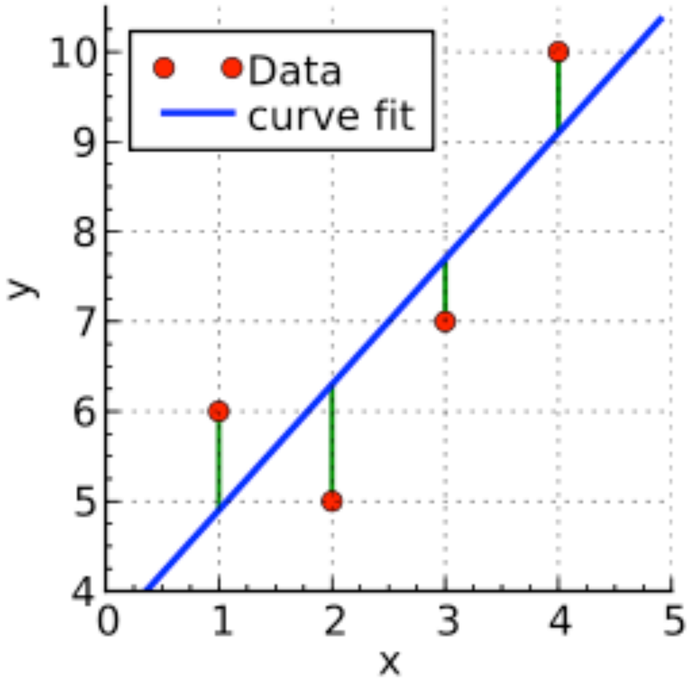


Sum of Square Errors *(MSE without the mean)*

$$y^i = \mathbf{w}^T \mathbf{x}^i + \epsilon^i$$

Loss function: sum of squared errors

$$L(\mathbf{w}) = \sum_{i=1}^{N} (\epsilon^i)^2$$



Sum of Square Errors *(MSE without the mean)*

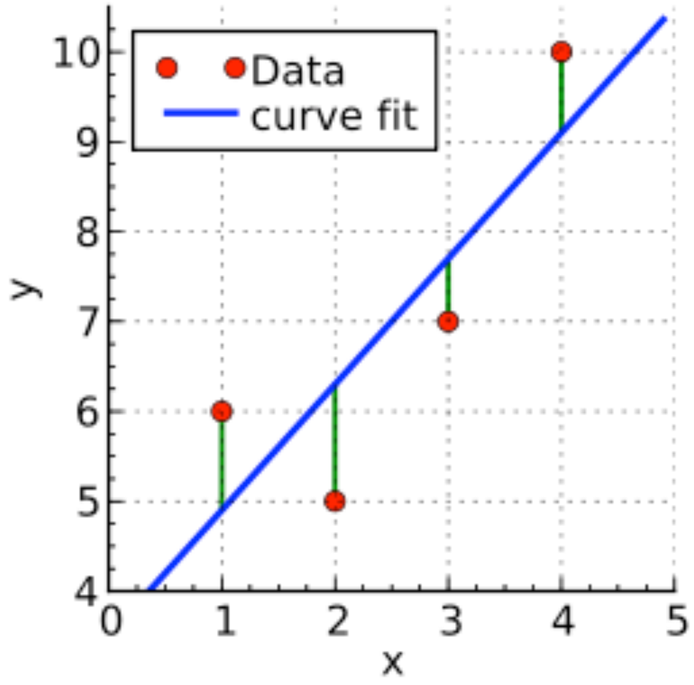
$$y^i = \mathbf{w}^T \mathbf{x}^i + \epsilon^i$$

Loss function: sum of squared errors

$$L(\mathbf{w}) = \sum_{i=1}^{N} (\epsilon^i)^2$$

In two variables:

$$L(w_0, w_1) = \sum_{i=1}^{N} \left[y^i - \left(w_0 x_0^i + w_1 x_1^i \right) \right]^2$$



Sum of Square Errors *(MSE without the mean)*

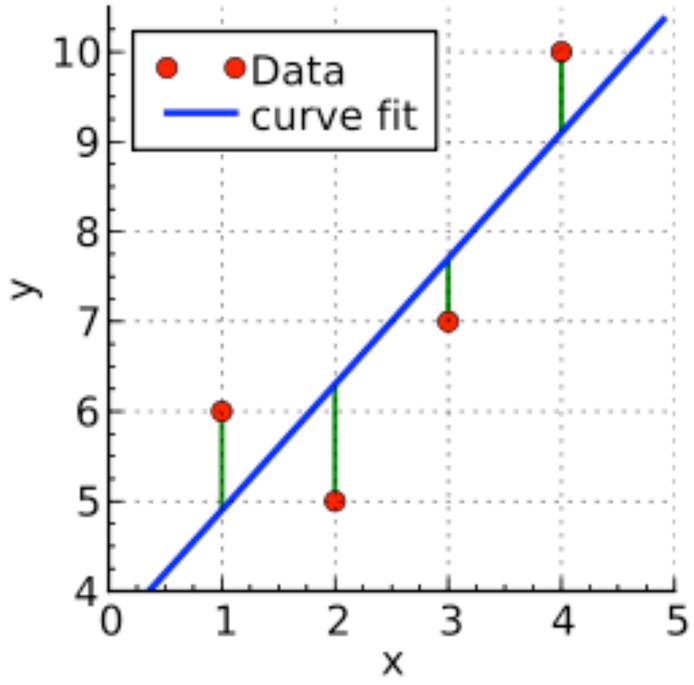
$$y^i = \mathbf{w}^T \mathbf{x}^i + \epsilon^i$$

Loss function: sum of squared errors

$$L(\mathbf{w}) = \sum_{i=1}^{N} (\epsilon^i)^2$$

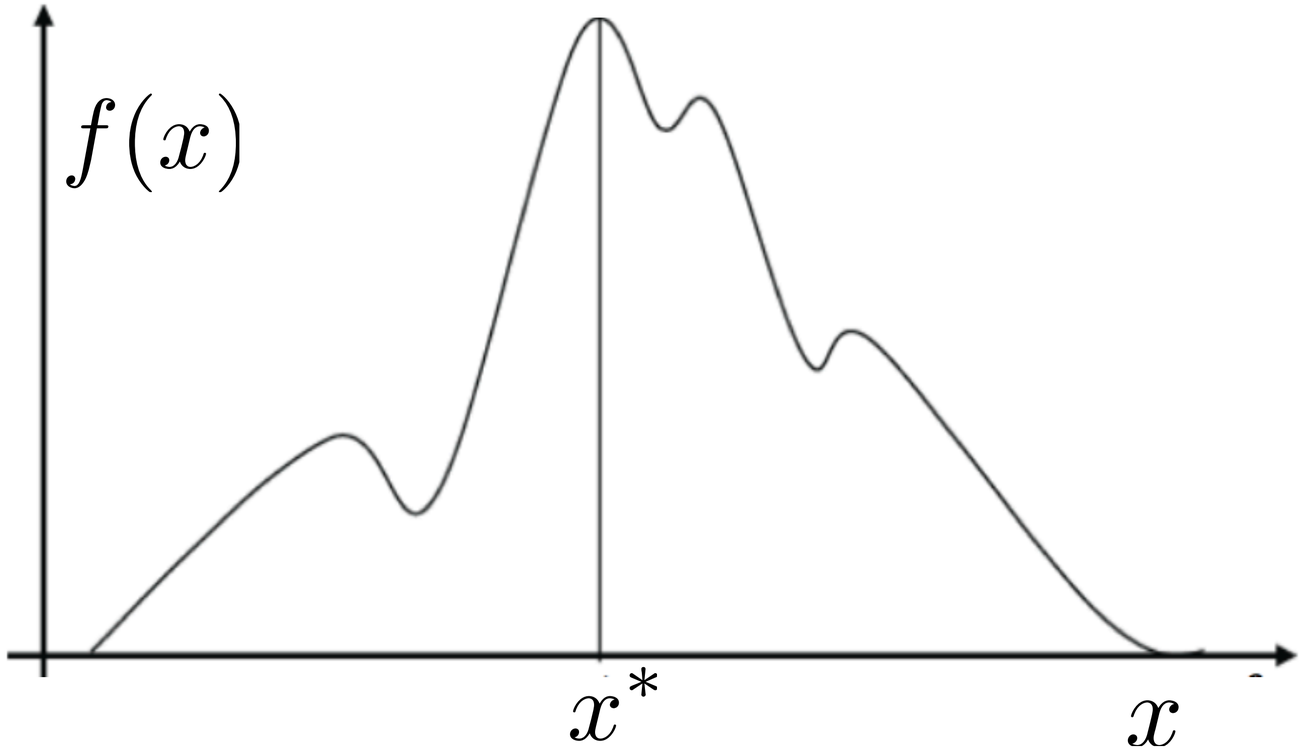
In two variables:

$$L(w_0, w_1) = \sum_{i=1}^{N} \left[y^i - \left(w_0 x_0^i + w_1 x_1^i \right) \right]^2$$

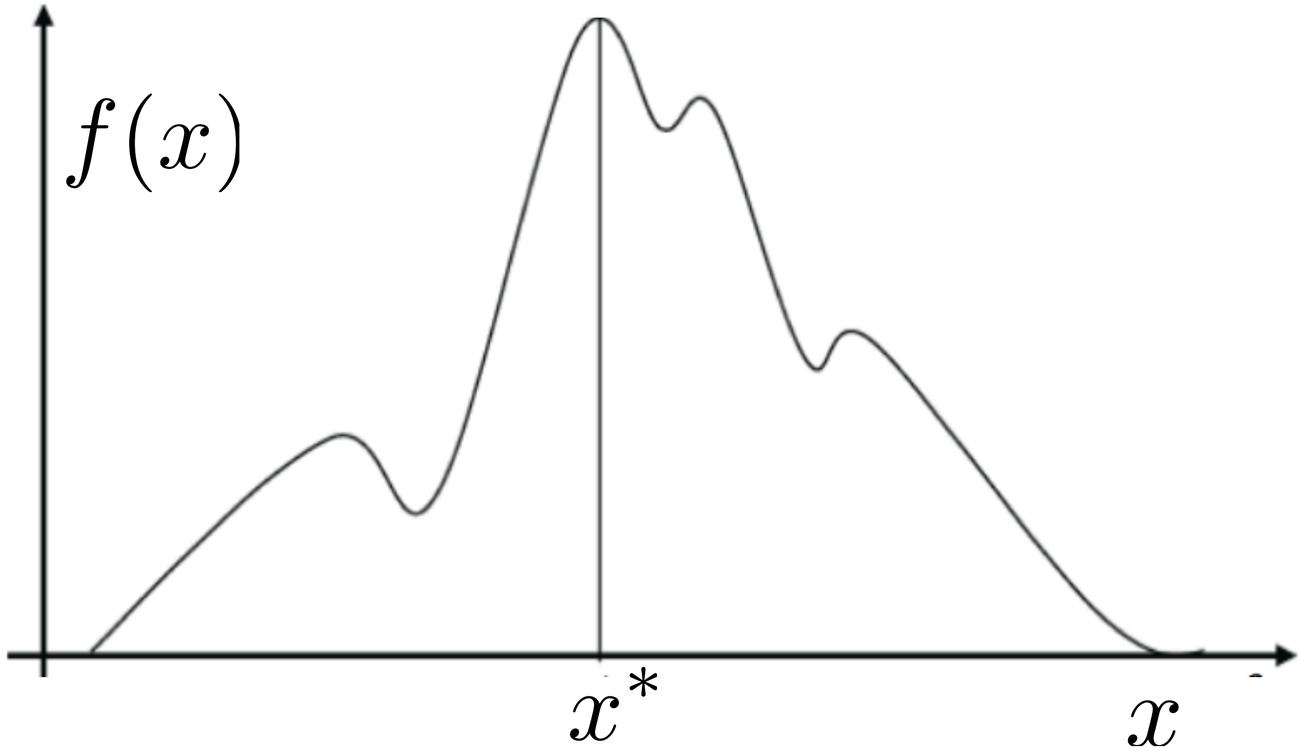


*Question: what is the best (or least bad) value of **w**?*

Calculus 101

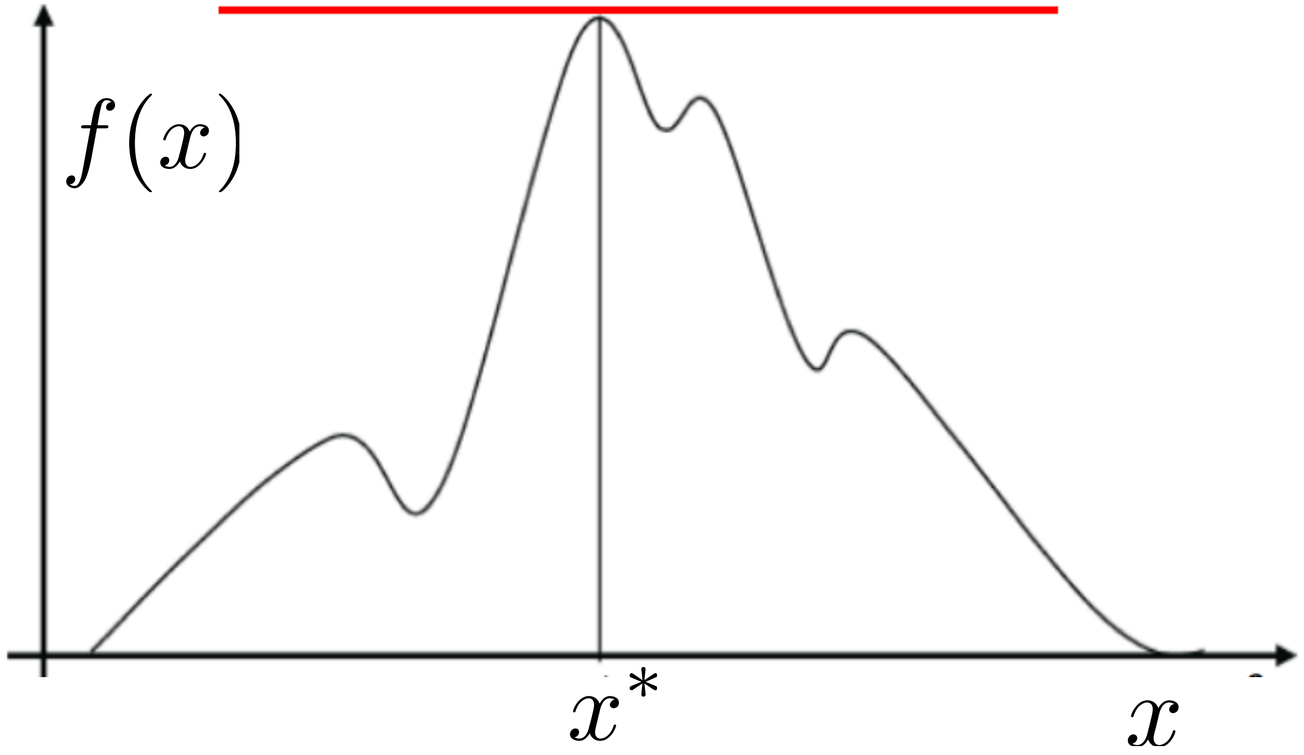


Calculus 101



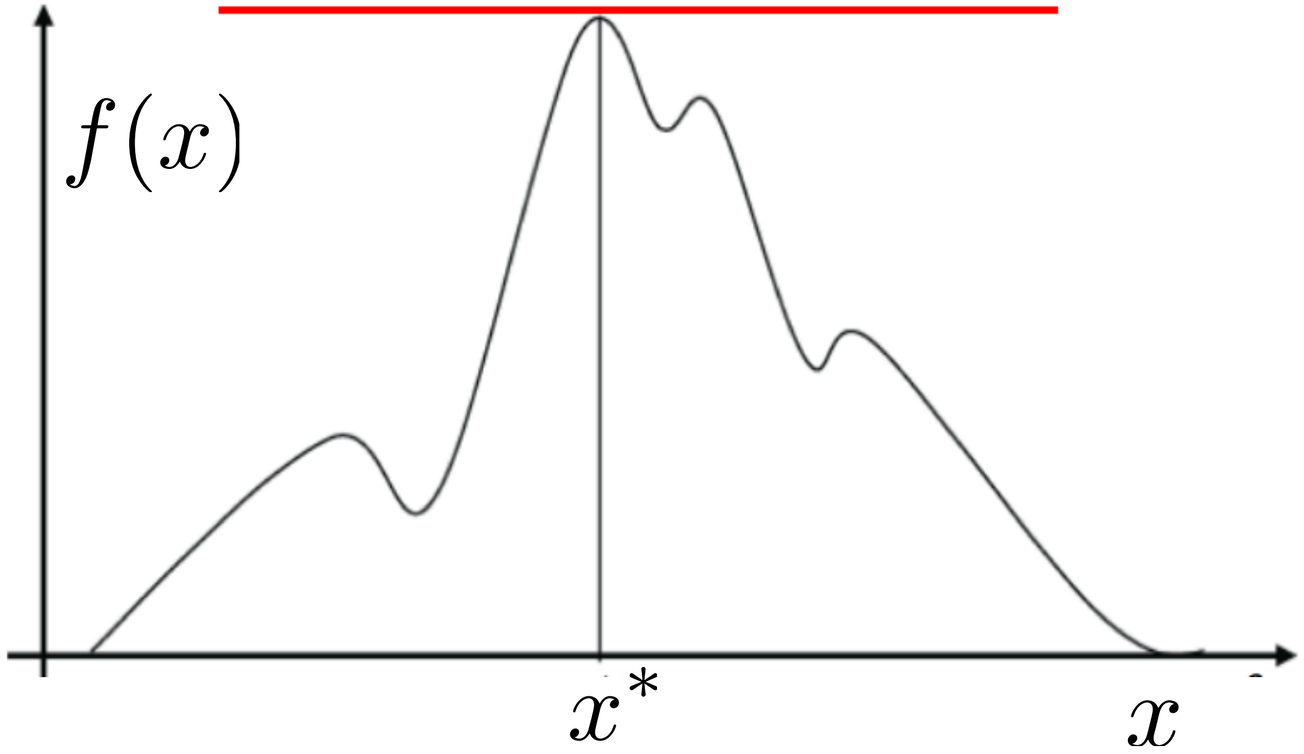
$$x^* = \text{argmax}_x f(x)$$

Local Extrema Condition



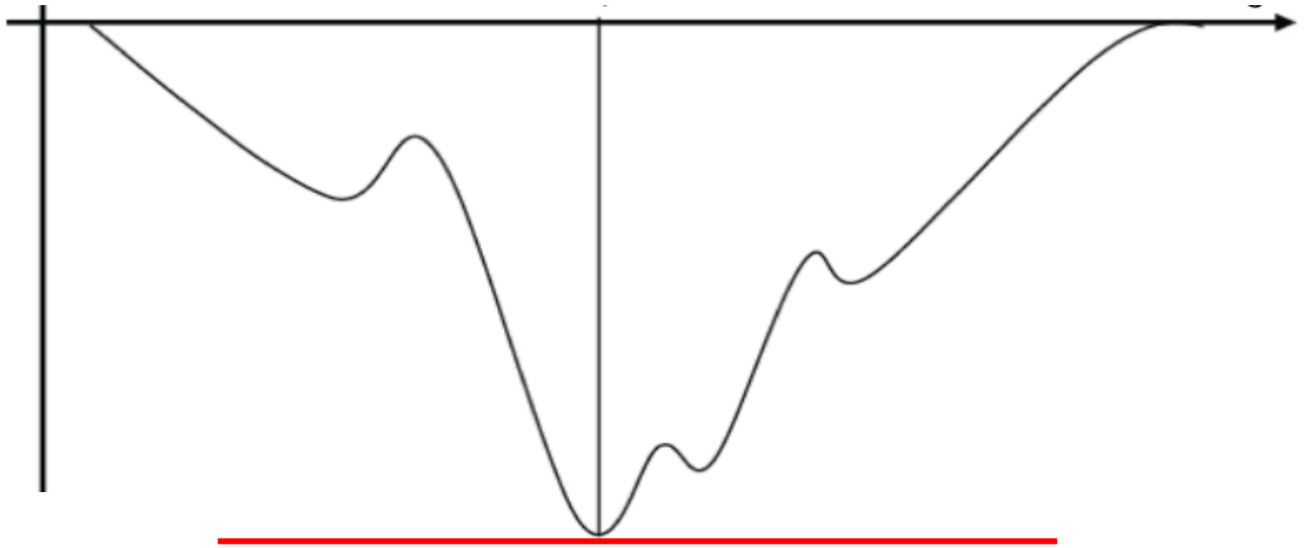
$$x^* = \text{argmax}_x f(x)$$

Local Extrema Condition



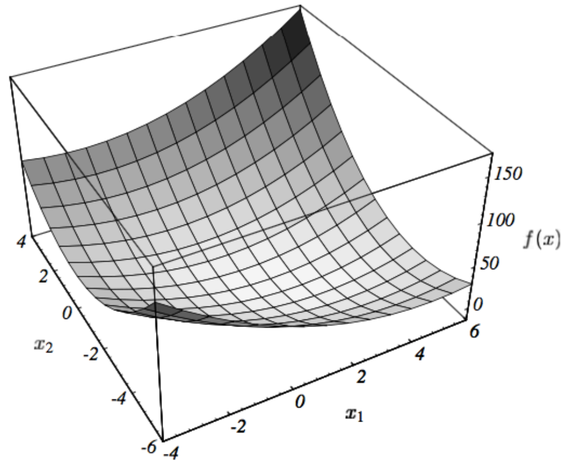
$$x^* = \text{argmax}_x f(x) \quad \rightarrow \quad f'(x^*) = 0$$

Local Extrema Condition



$$x^* = \text{argmax}_x f(x) \quad \rightarrow \quad f'(x^*) = 0$$

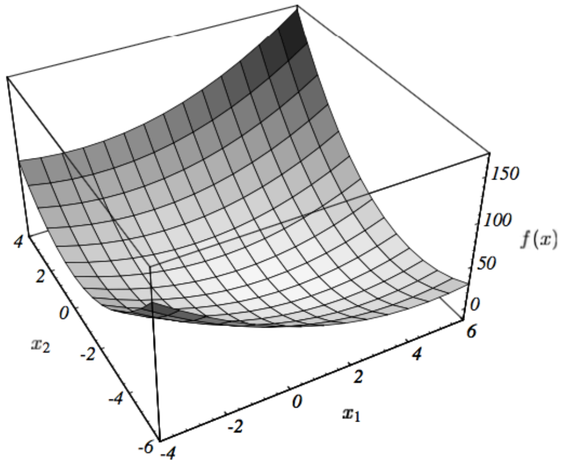
Vector Calculus 101

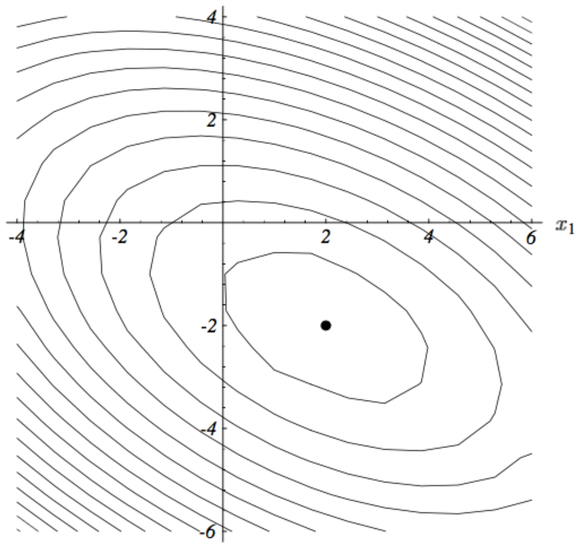


$$f(\mathbf{x})$$

2D function graph

Vector Calculus 101





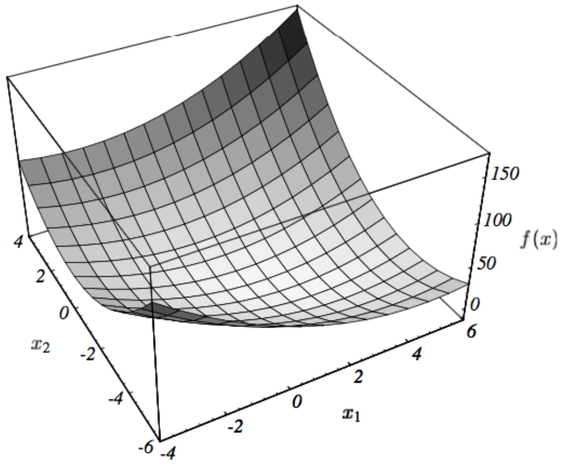
$$f(\mathbf{x})$$

$$f(\mathbf{x}) = c$$

2D function graph

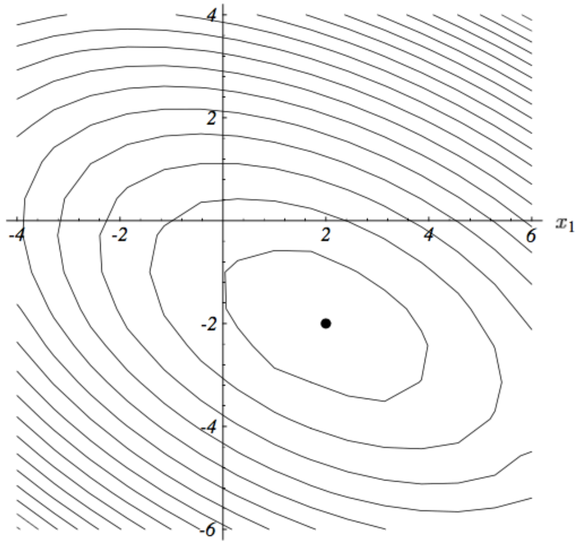
isocontours

Vector Calculus 101



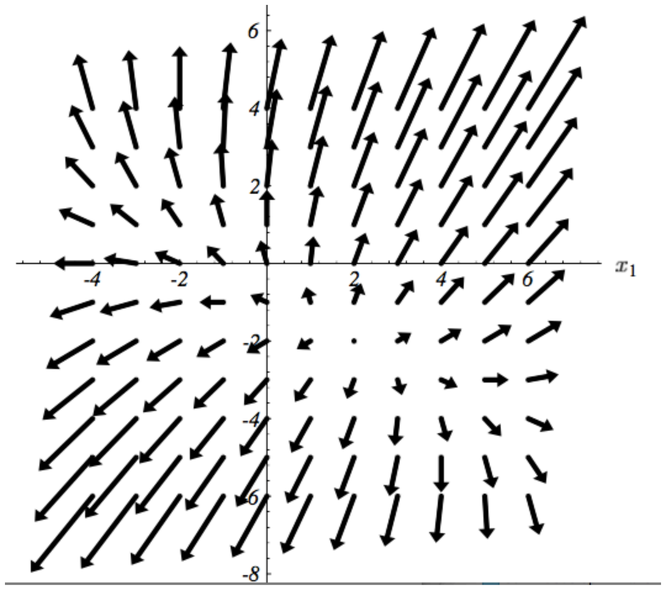
$$f(\mathbf{x})$$

2D function graph



$$f(\mathbf{x}) = c$$

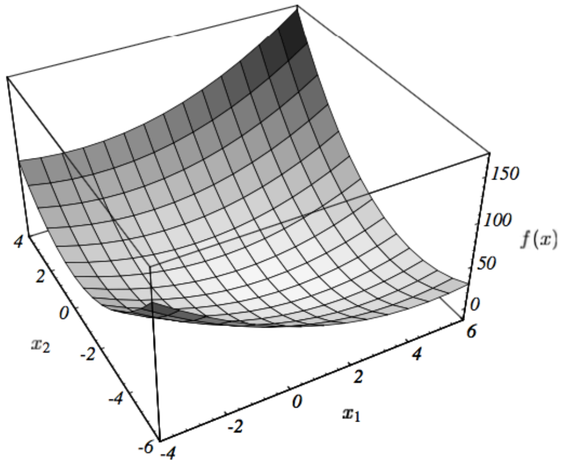
isocontours

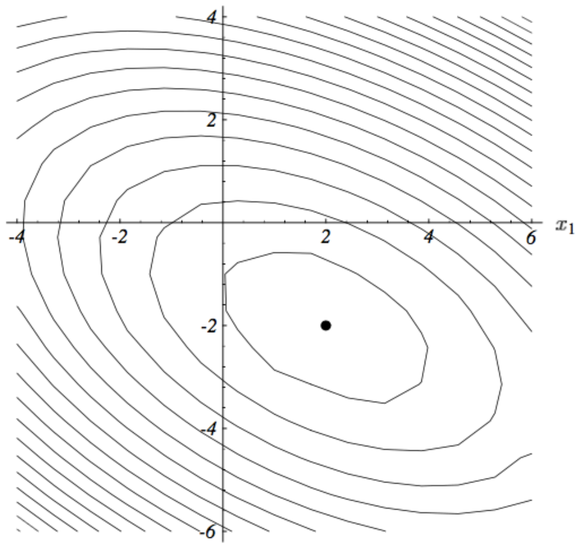


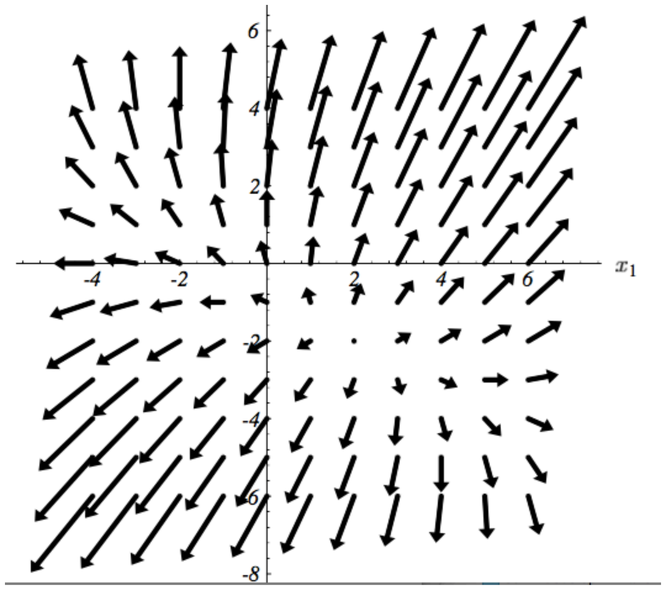
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

gradient field

Vector Calculus 101







$$f(\mathbf{x})$$

$$f(\mathbf{x}) = c$$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

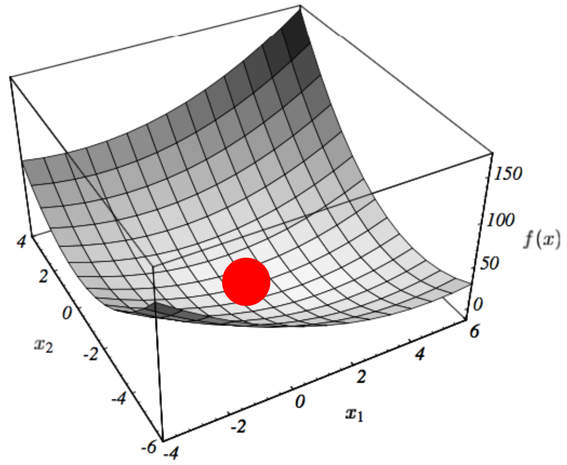
2D function graph

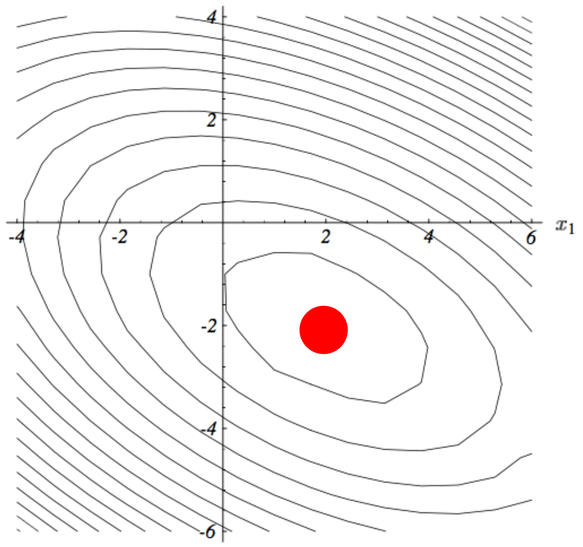
isocontours

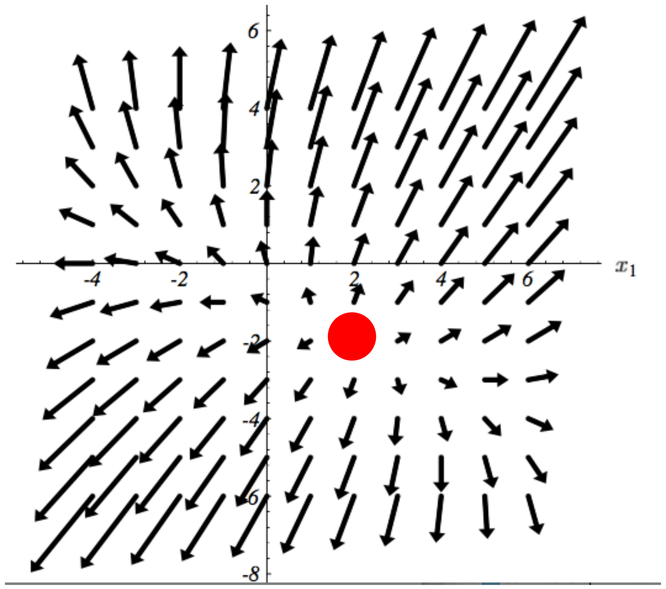
gradient field

- at minimum of function: $\nabla f(\mathbf{x}) = \mathbf{0}$

Vector Calculus 101







$f(\mathbf{x})$

$f(\mathbf{x}) = c$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

2D function graph

isocontours

gradient field

- at minimum of function: $\nabla f(\mathbf{x}) = \mathbf{0}$

LS Solution for Linear Regression

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$$

$$L(\mathbf{w}) = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon}$$

$$\mathbf{X} = \begin{bmatrix} (\mathbf{x}^1)^T \\ \hline (\mathbf{x}^2)^T \\ \hline \vdots \\ \hline (\mathbf{x}^N)^T \end{bmatrix}$$

LS Solution for Linear Regression

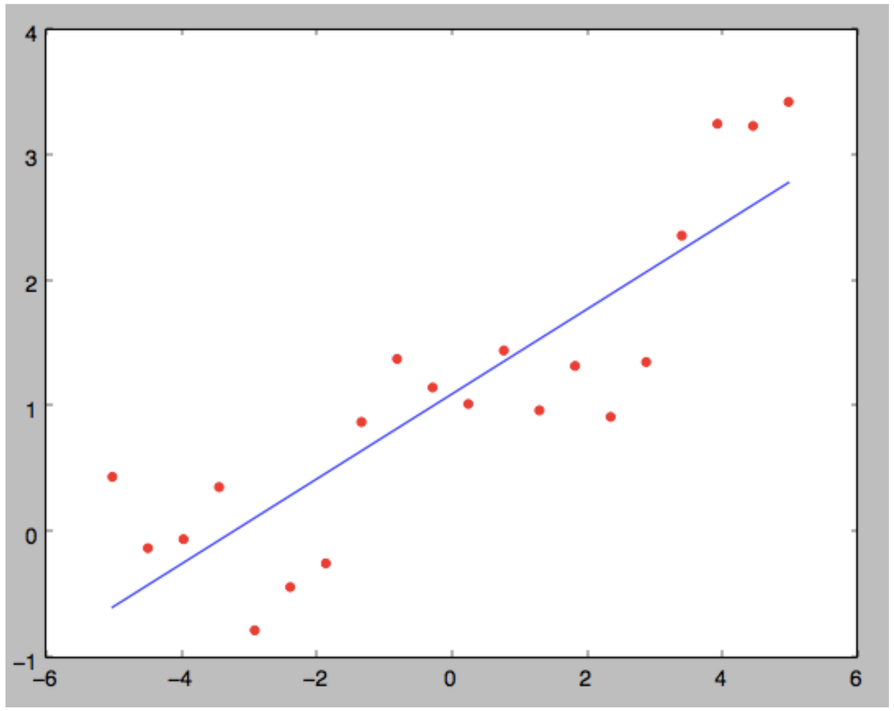
$$\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$$

$$L(\mathbf{w}) = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon}$$

$$\mathbf{X} = \begin{bmatrix} \frac{(\mathbf{x}^1)^T}{(\mathbf{x}^2)^T} \\ \vdots \\ \frac{}{(\mathbf{x}^N)^T} \end{bmatrix}$$

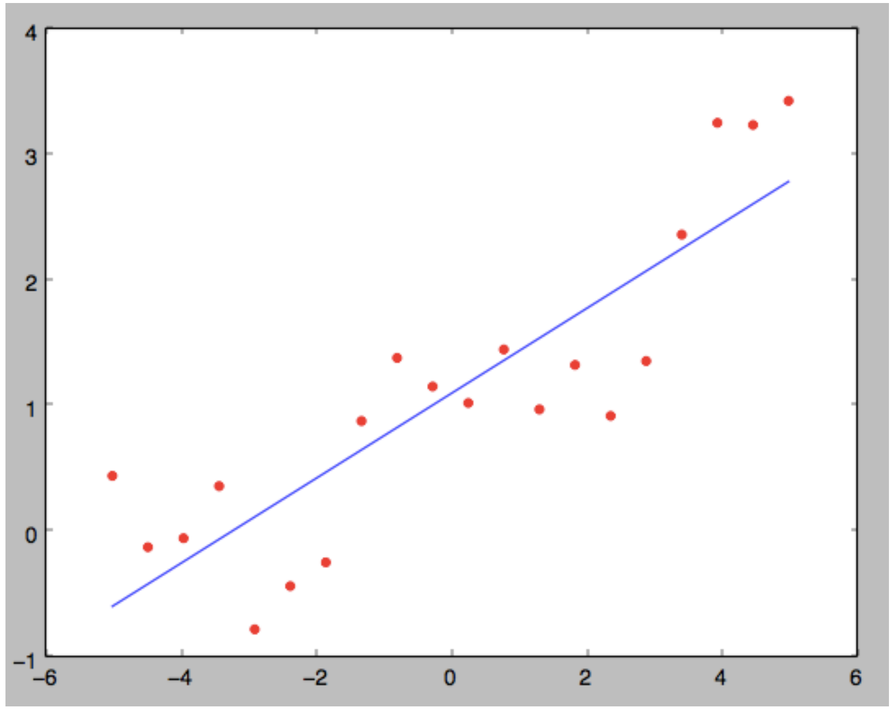
$$\mathbf{w}^* \leftarrow (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Code Example



$$\mathbf{w}^* \leftarrow (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

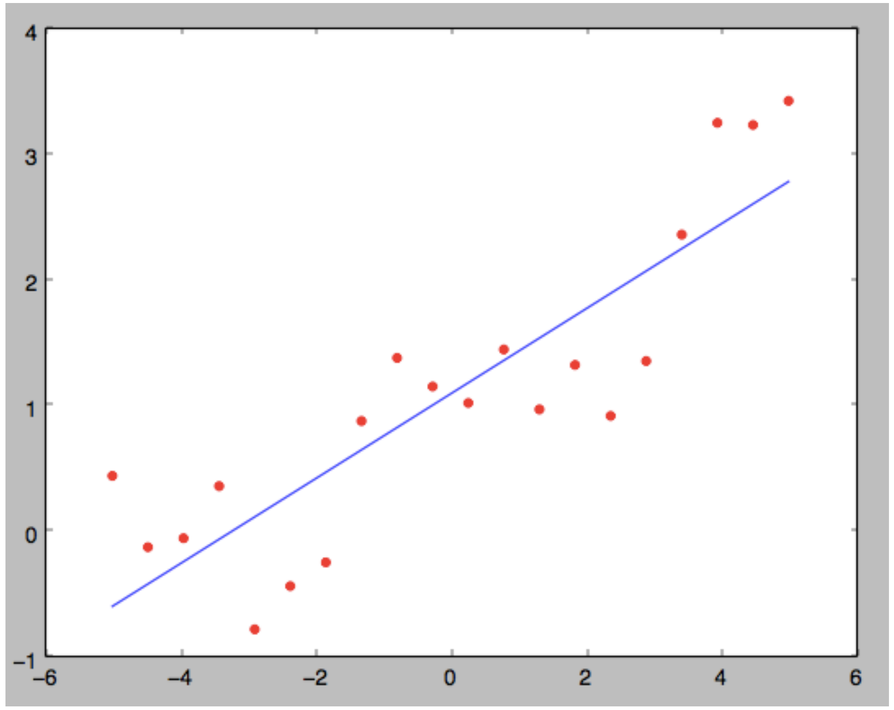
Code Example




```python
import numpy as np
from numpy import array
from numpy import matmul
from numpy.linalg import inv
from numpy.random import rand
from matplotlib import pyplot

# generate data on a line perturbed with some noise
noise_margin= 2
w = rand(2,1) # w[0] is random constant term (offset from origin), w[1] is random linear term (slope)
x  = np.linspace(-5,5,20)
y = w[0] + w[1]*x + noise_margin*rand(len(x))

# create the design matrix: the x data, and add a column of ones for the constant term
X = np.column_stack( [np.ones([len(x), 1]), x.reshape(-1, 1)] )

# These are the normal equations in matrix form: w = (X' X)^-1 X' y
w_est = matmul(inv(matmul(X.transpose(),X)),X.transpose()).dot(y)

# For ridge regression, use regularizer
#weight = 0.01
#w_est = matmul(inv(matmul(X.transpose(),X) + weight*np.identity(2)),X.transpose()).dot(y)


# evaluate the x values in the fitted model to get estimated y values
y_est = w_est[0] + w_est[1]*x

# visualize the fitted model
pyplot.scatter(x, y, color='red')
pyplot.plot(x, y_est, color='blue')
pyplot.show()
```

$$\mathbf{w}^* \leftarrow (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

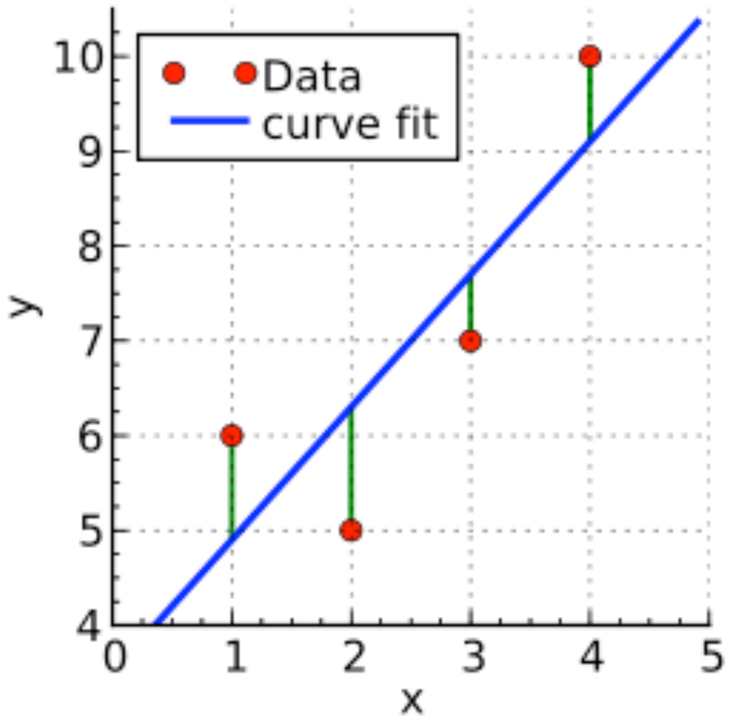
Code Example



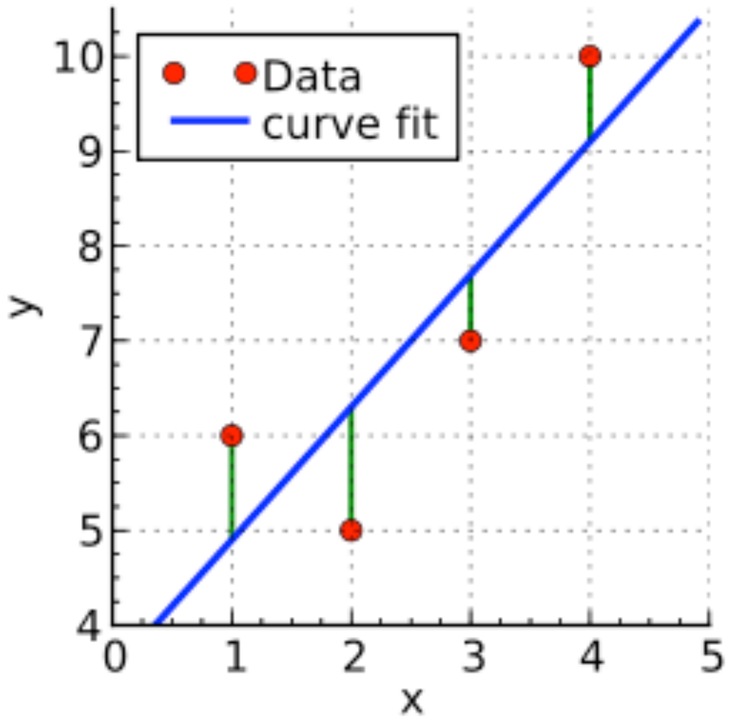

```python
import numpy as np
from numpy import array
from numpy import matmul
from numpy.linalg import inv
from numpy.random import rand
from matplotlib import pyplot

# generate data on a line perturbed with some noise
noise_margin= 2
w = rand(2,1) # w[0] is random constant term (offset from origin), w[1] is random linear term (slope)
x  = np.linspace(-5,5,20)
y = w[0] + w[1]*x + noise_margin*rand(len(x))

# create the design matrix: the x data, and add a column of ones for the constant term
X = np.column_stack( [np.ones([len(x), 1]), x.reshape(-1, 1)] )

# These are the normal equations in matrix form: w = (X' X)^-1 X' y
w_est = matmul(inv(matmul(X.transpose(),X)),X.transpose()).dot(y)

# For ridge regression, use regularizer
#weight = 0.01
#w_est = matmul(inv(matmul(X.transpose(),X) + weight*np.identity(2)),X.transpose()).dot(y)


# evaluate the x values in the fitted model to get estimated y values
y_est = w_est[0] + w_est[1]*x

# visualize the fitted model
pyplot.scatter(x, y, color='red')
pyplot.plot(x, y_est, color='blue')
pyplot.show()
```

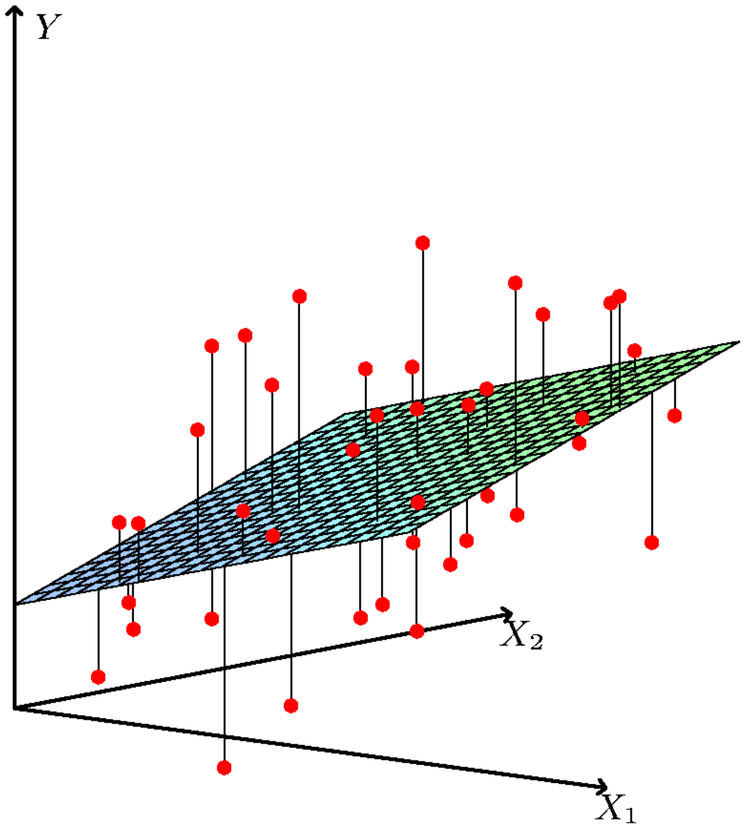
$$\mathbf{w}^* \leftarrow (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Linear Regression (Line/Plane Fitting)



Linear Regression (Line/Plane Fitting)





LS Solution for Regression

$$L(\mathbf{w}) = \sum_{i=1}^{N} (y^i - \mathbf{w}^T \mathbf{x}^i)^2 = \sum_{i=1}^{N} (\epsilon^i)^2$$

$$L(\mathbf{w}) = \begin{bmatrix} \epsilon^1 & \epsilon^2 & \dots & \epsilon^N \end{bmatrix} \begin{bmatrix} \epsilon^1 \\ \epsilon^2 \\ \vdots \\ \epsilon^N \end{bmatrix}$$

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$$

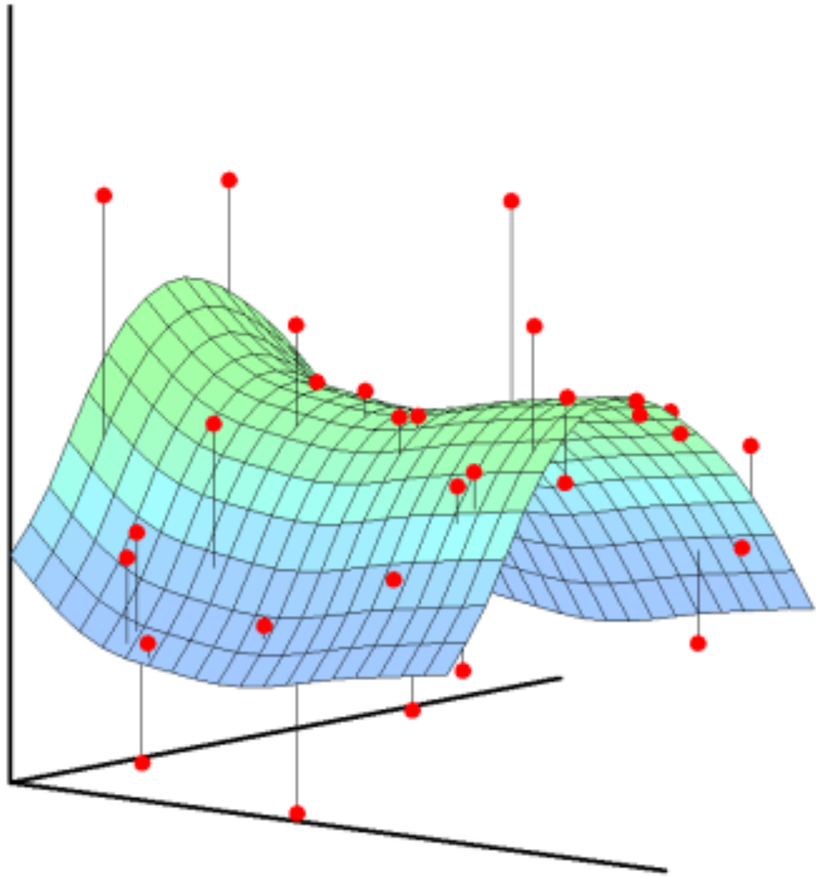
LS Solution for Regression

$$L(\mathbf{w}) = \sum_{i=1}^{N} (y^i - \mathbf{w}^T \mathbf{x}^i)^2 = \sum_{i=1}^{N} (\epsilon^i)^2$$

$$L(\mathbf{w}) = \begin{bmatrix} \epsilon^1 & \epsilon^2 & \dots & \epsilon^N \end{bmatrix} \begin{bmatrix} \epsilon^1 \\ \epsilon^2 \\ \vdots \\ \epsilon^N \end{bmatrix}$$

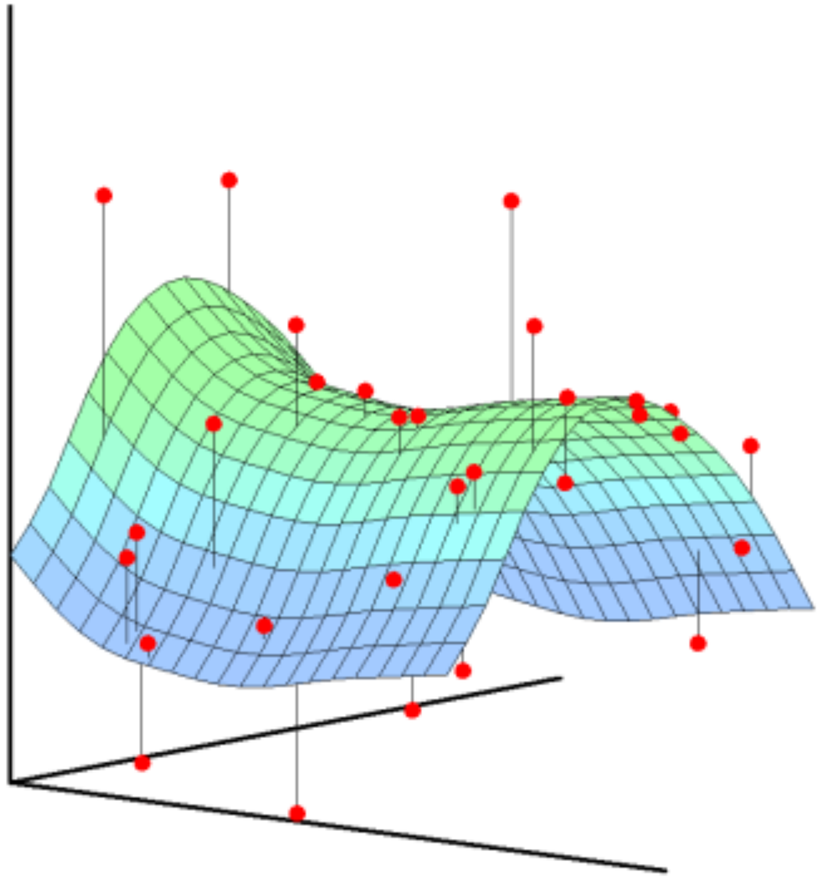
$$L(\mathbf{w}) = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon} \qquad \mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$$

Generalized Linear Regression



$$\mathbf{x} \rightarrow \boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} \phi_1(\mathbf{x}) \\ \vdots \\ \phi_M(\mathbf{x}) \end{bmatrix}$$

Generalized Linear Regression

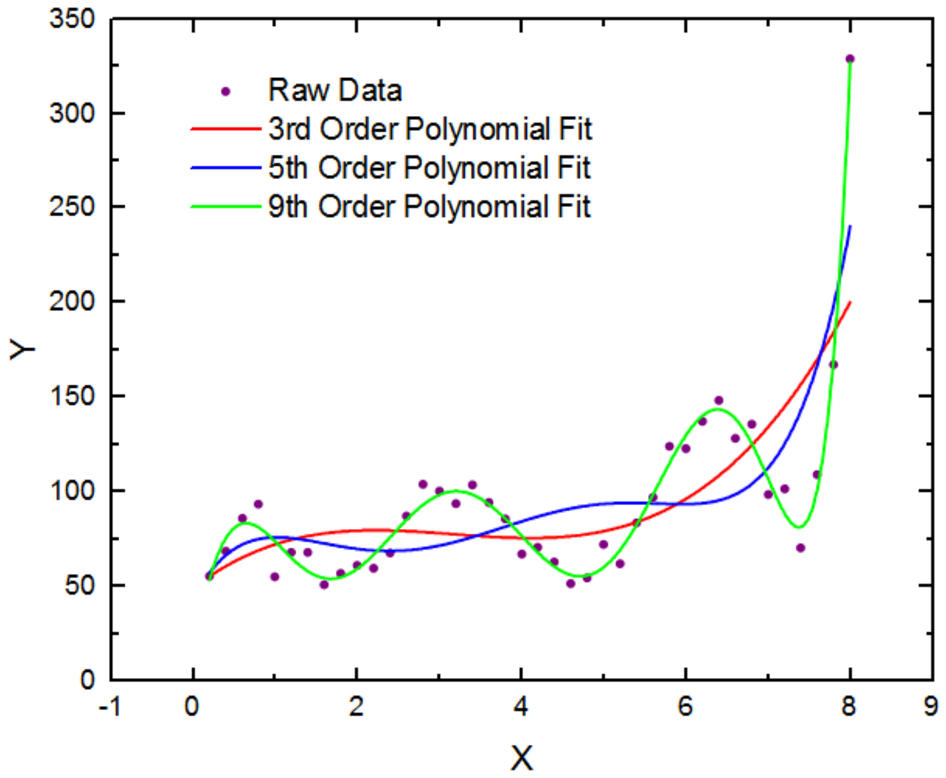


$$\mathbf{x} \rightarrow \boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} \phi_1(\mathbf{x}) \\ \vdots \\ \phi_M(\mathbf{x}) \end{bmatrix}$$

known nonlinearity

1D Example: k-th Degree Polynomial Fitting

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} 1 \\ x \\ \vdots \\ (x)^K \end{bmatrix}$$



$$\langle \mathbf{w}, \boldsymbol{\phi}(x) \rangle = w_0 + w_1 x + \ldots + w_k (x)^K$$

Generalized Linear Regression

$$L(\mathbf{w}) = \sum_{i=1}^{N} (y^i - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}^i))^T = \sum_{i=1}^{N} (\epsilon^i)^2$$

Generalized Linear Regression

$$L(\mathbf{w}) = \sum_{i=1}^{N} (y^i - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}^i))^T = \sum_{i=1}^{N} (\epsilon^i)^2$$

$$\underset{\text{Nx1}}{\begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^N \end{bmatrix}} = \underset{\text{NxM}}{\begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}^1)^T \\ \hline \boldsymbol{\phi}(\mathbf{x}^2)^T \\ \hline \vdots \\ \hline \boldsymbol{\phi}(\mathbf{x}^N)^T \end{bmatrix}} \underset{\text{Mx1}}{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix}} + \underset{\text{Nx1}}{\begin{bmatrix} \epsilon^1 \\ \epsilon^2 \\ \vdots \\ \epsilon^N \end{bmatrix}}$$

$$\boldsymbol{\phi}(\mathbf{x}) : \mathbb{R}^D \rightarrow \mathbb{R}^M$$

LS Solution for Generalized Linear Regression

$$\mathbf{y} = \mathbf{\Phi w} + \boldsymbol{\epsilon}$$

$$\mathbf{\Phi} = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}^1)^T \\ \hline \boldsymbol{\phi}(\mathbf{x}^2)^T \\ \hline \vdots \\ \hline \boldsymbol{\phi}(\mathbf{x}^N)^T \end{bmatrix}$$

LS Solution for Generalized Linear Regression

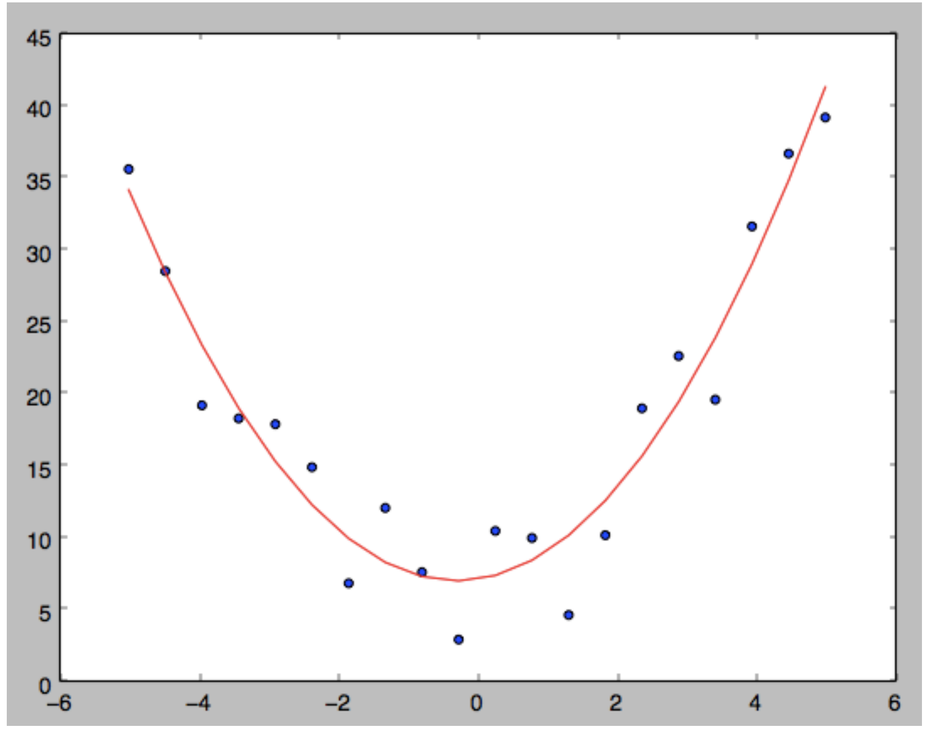
$$\mathbf{y} = \mathbf{\Phi w} + \boldsymbol{\epsilon}$$

$$L(\mathbf{w}) = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon}$$

$$\mathbf{\Phi} = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}^1)^T \\ \hline \boldsymbol{\phi}(\mathbf{x}^2)^T \\ \hline \vdots \\ \hline \boldsymbol{\phi}(\mathbf{x}^N)^T \end{bmatrix}$$

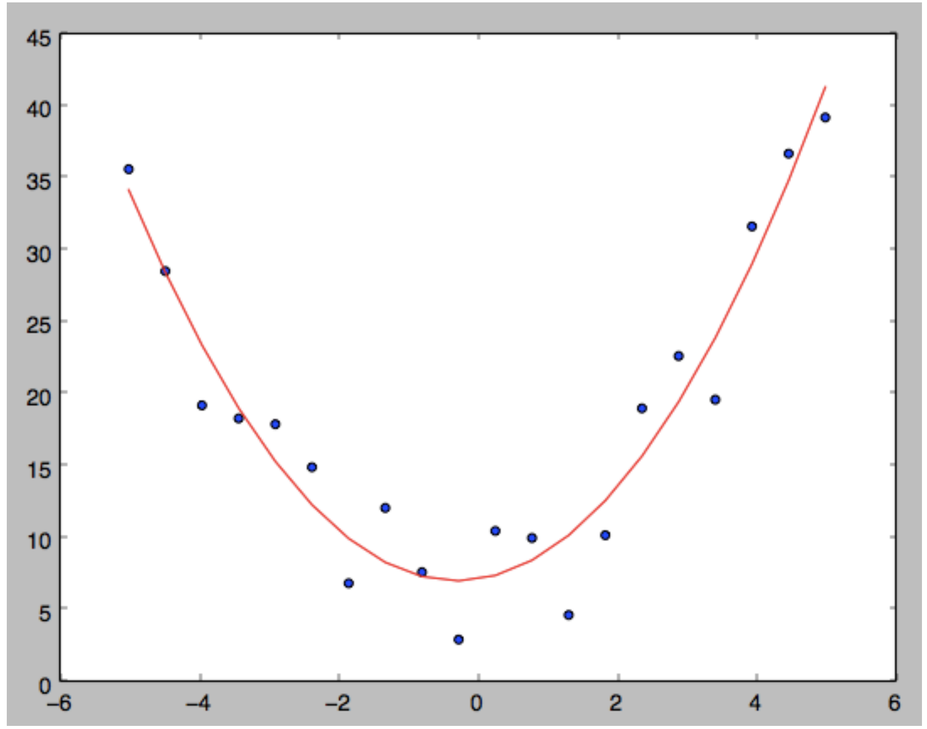
$$\mathbf{w}^* = (\mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi y}$$

Code Example




```
import numpy as np
from numpy import array
from numpy import matmul
from numpy.linalg import inv
from numpy.random import rand
from matplotlib import pyplot

# generate data on a line perturbed with some noise
noise_margin= 3
w = 2*rand(3,1) # w[0] is random constant term (offset from origin), w[1] is random linear term, w[2] is random quadratic term
x  = np.linspace(-5,5,20)
y = w[0] + w[1]*x + w[2]*x**2 + noise_margin*rand(len(x))

# create the design matrix: the x data, and add a column of ones for the constant term
X = np.column_stack( [np.ones([len(x), 1]), x.reshape(-1, 1), (x**2).reshape(-1, 1)] )

# These are the normal equations in matrix form: w = (X' X)^-1 X' y
w_est = matmul(inv(matmul(X.transpose(),X)),X.transpose()).dot(y)


# evaluate the x values in the fitted model to get estimated y values
y_est = w_est[0] + w_est[1]*x + w_est[2]*x**2

# visualize the fitted model
pyplot.scatter(x, y)
pyplot.plot(x, y_est, color='red')
pyplot.show()
```

Code Example



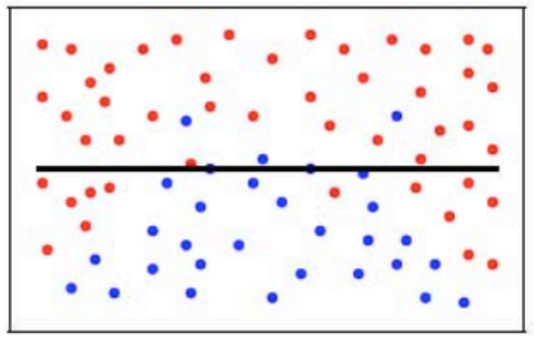

```
import numpy as np
from numpy import array
from numpy import matmul
from numpy.linalg import inv
from numpy.random import rand
from matplotlib import pyplot

# generate data on a line perturbed with some noise
noise_margin= 3
w = 2*rand(3,1) # w[0] is random constant term (offset from origin), w[1] is random linear term, w[2] is random quadratic term
x  = np.linspace(-5,5,20)
y = w[0] + w[1]*x + w[2]*x**2 + noise_margin*rand(len(x))

# create the design matrix: the x data, and add a column of ones for the constant term
X = np.column_stack( [np.ones([len(x), 1]), x.reshape(-1, 1), (x**2).reshape(-1, 1)] )

# These are the normal equations in matrix form: w = (X' X)^-1 X' y
w_est = matmul(inv(matmul(X.transpose(),X)),X.transpose()).dot(y)


# evaluate the x values in the fitted model to get estimated y values
y_est = w_est[0] + w_est[1]*x + w_est[2]*x**2

# visualize the fitted model
pyplot.scatter(x, y)
pyplot.plot(x, y_est, color='red')
pyplot.show()
```

$$\mathbf{w}^* = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}\mathbf{y}$$

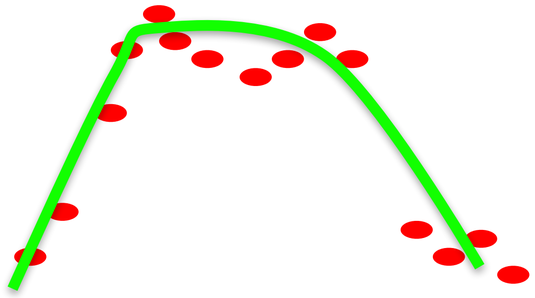
Hyperparameter: Underfitting vs. Overfitting

Underfitting

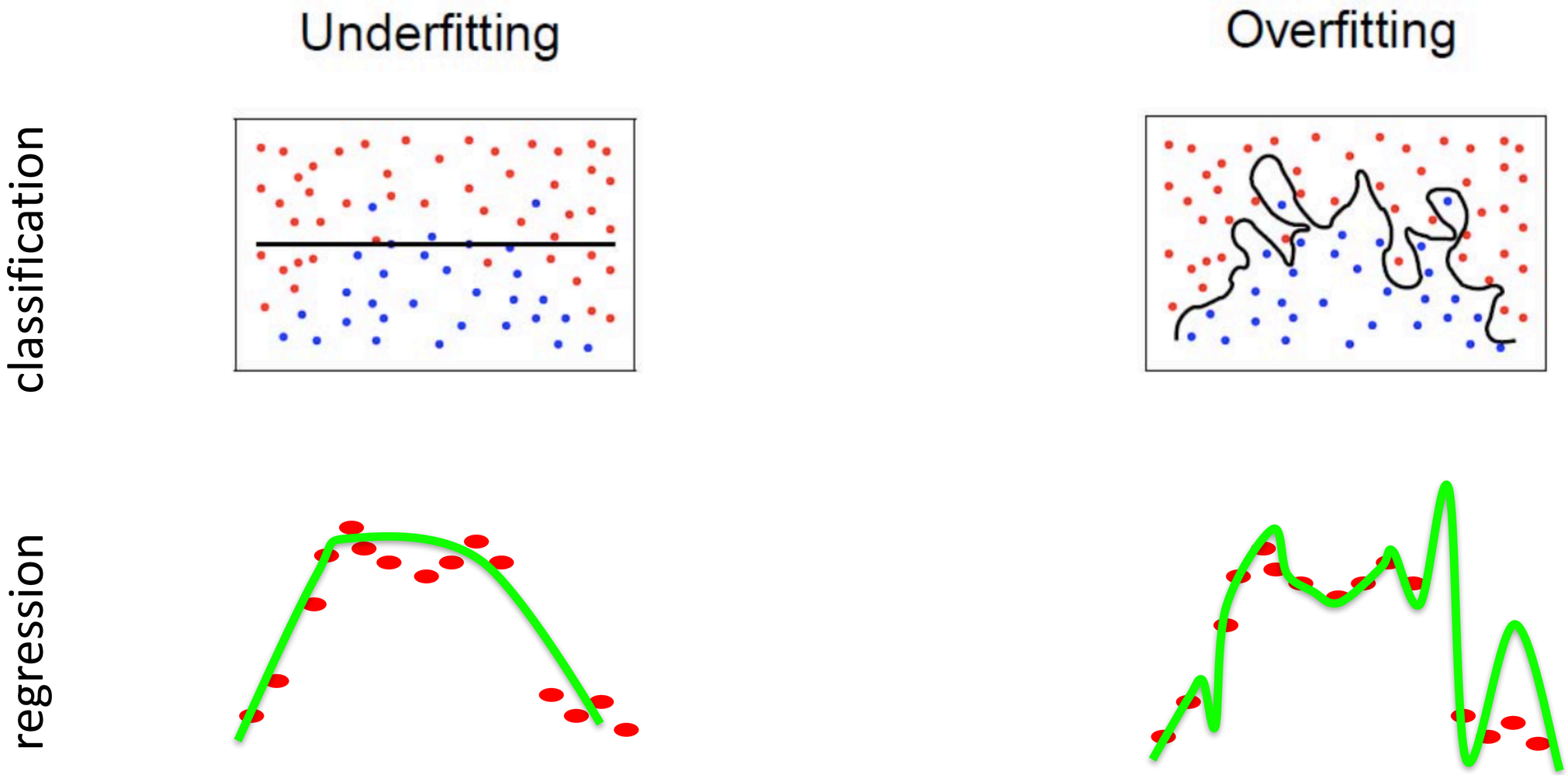
classification



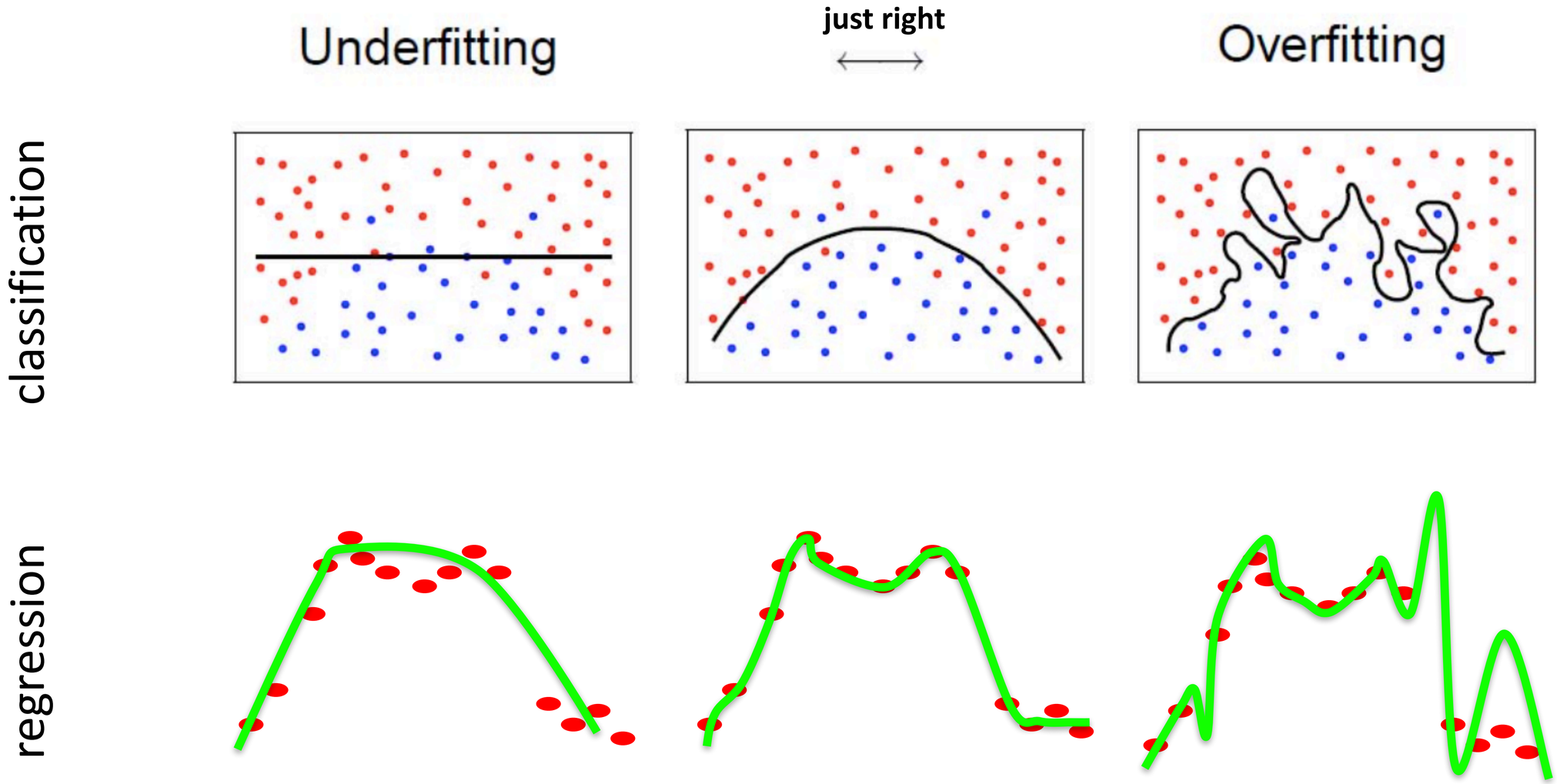
regression



Hyperparameter: Underfitting vs. Overfitting

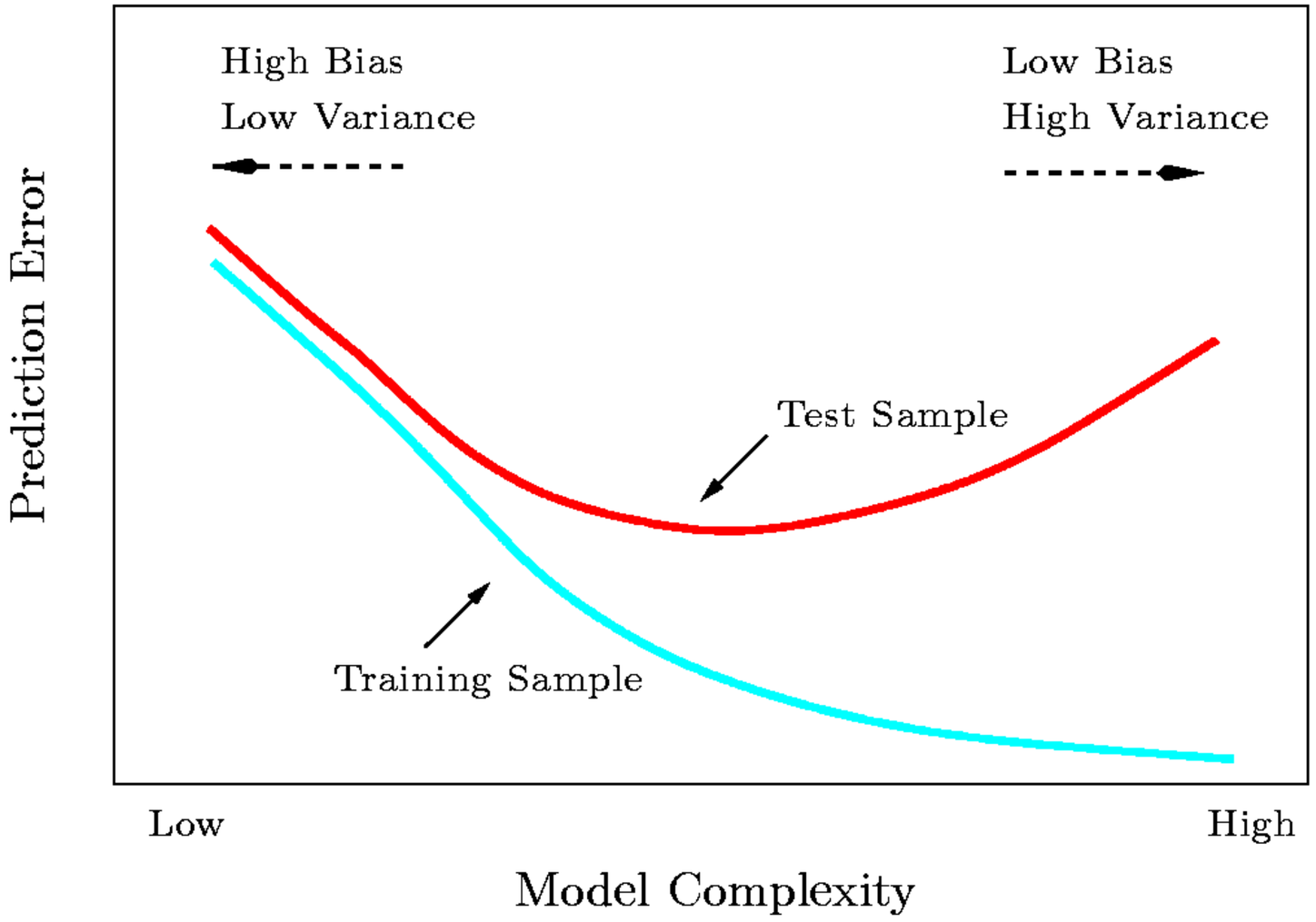


Hyperparameter: Underfitting vs. Overfitting

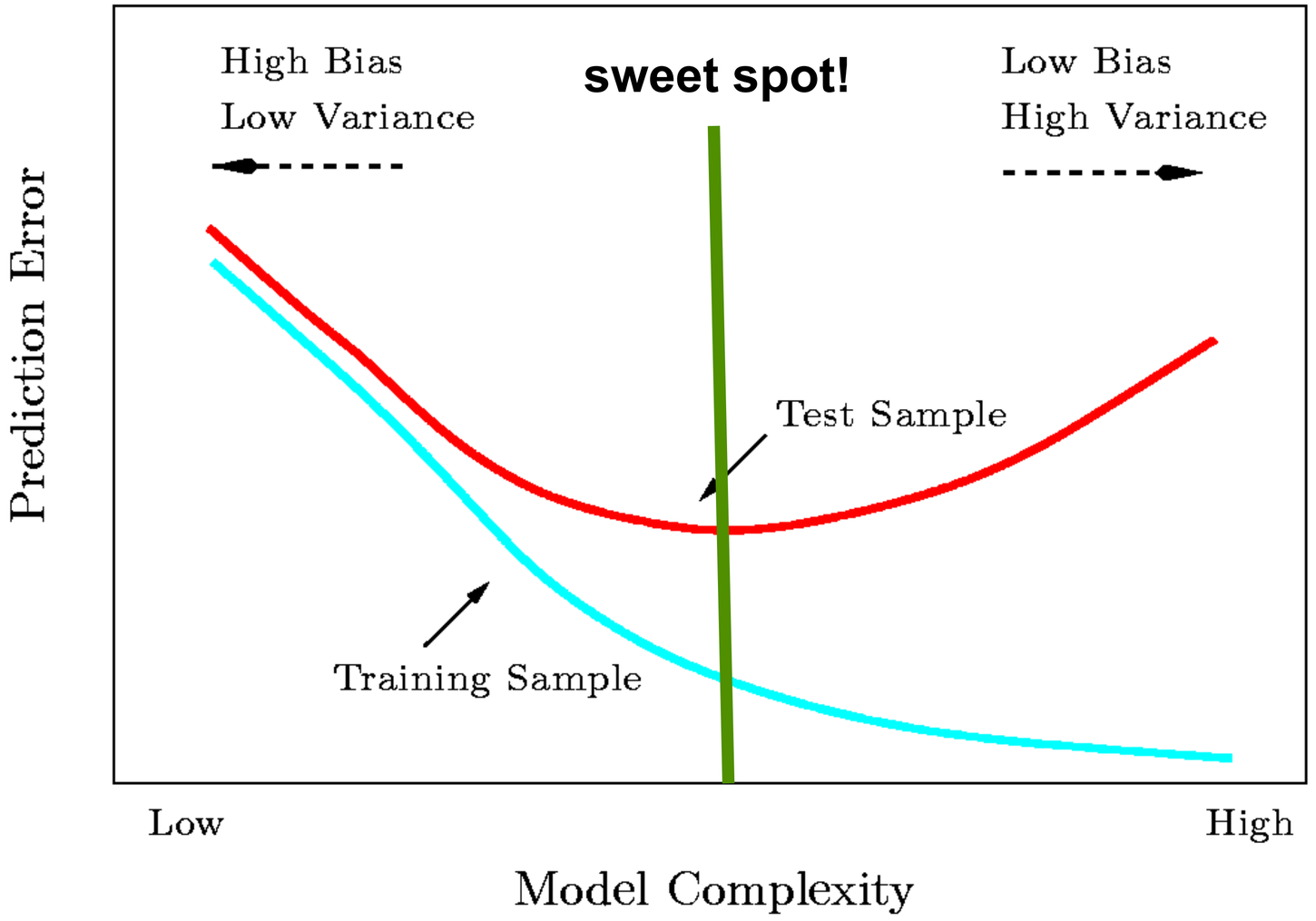


Hyperparameter Tuning

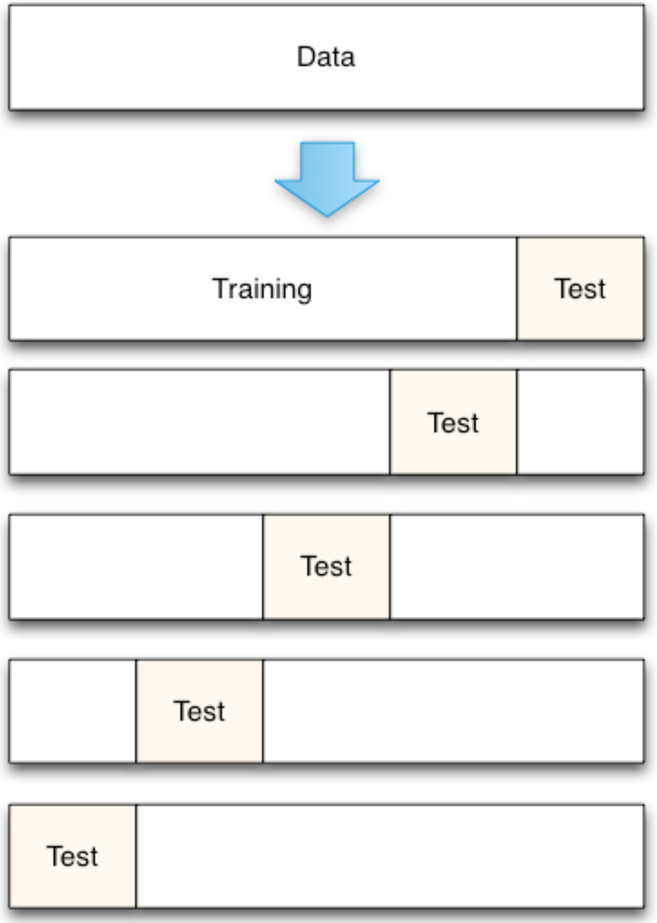
Hyperparameter Tuning



Hyperparameter Tuning

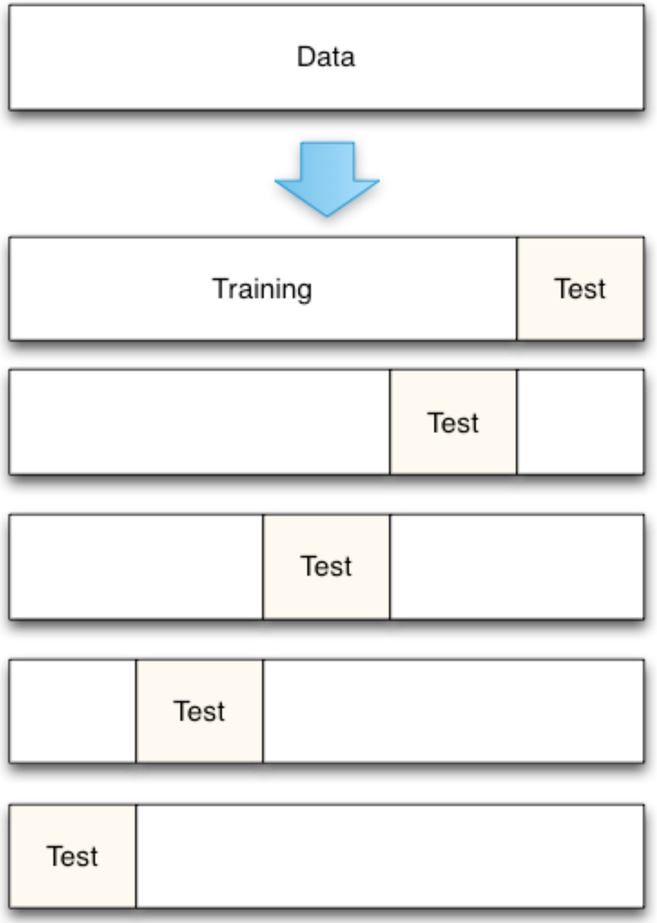


Selecting λ with Cross-validation



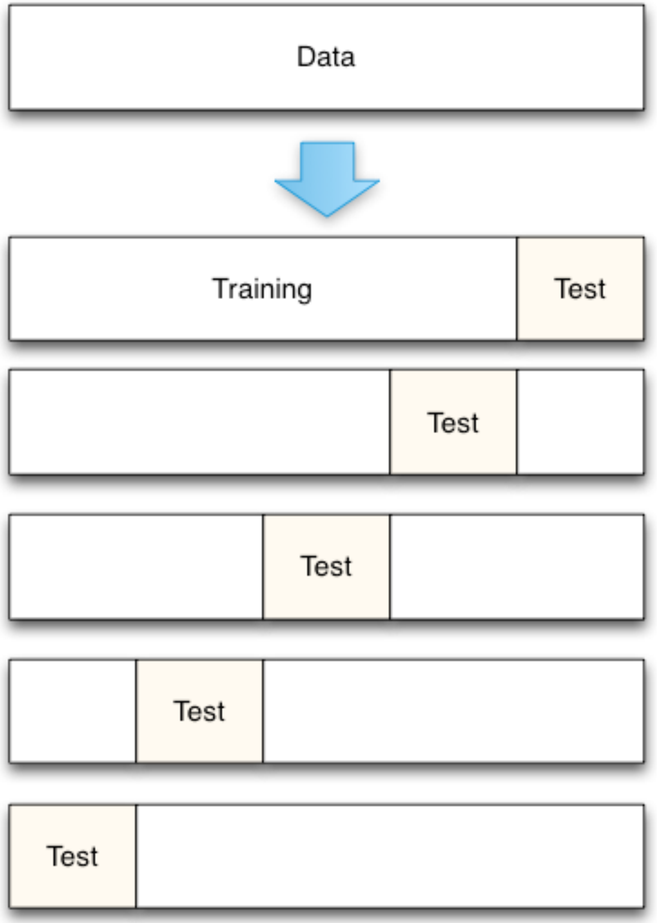
Selecting λ with Cross-validation

- Cross validation technique
  - Exclude part of the training data from parameter estimation
  - Use them only to predict the test error



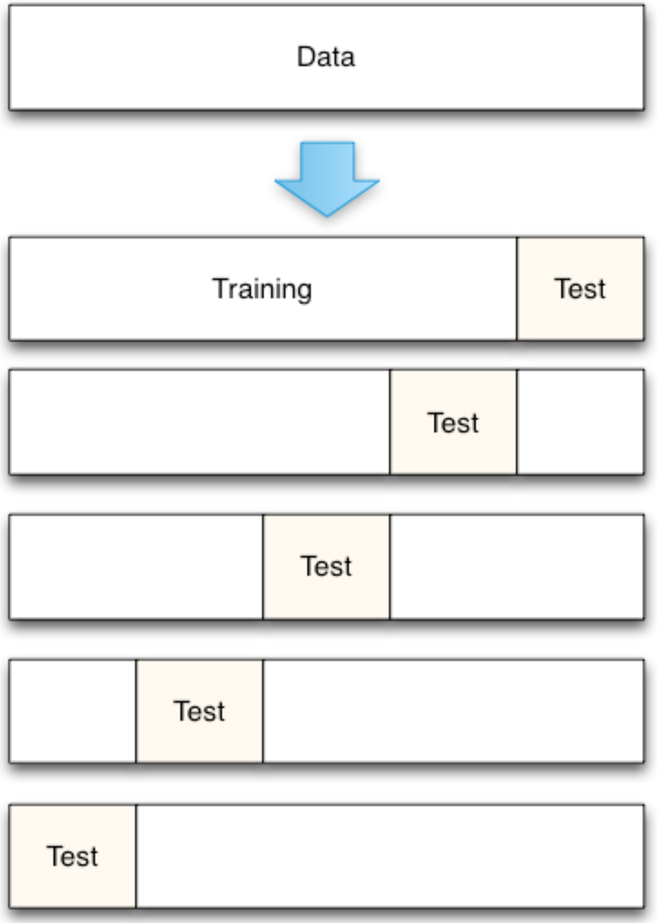
Selecting λ with Cross-validation

- Cross validation technique
  - Exclude part of the training data from parameter estimation
  - Use them only to predict the test error
- K-fold cross validation:
  - K splits, average K errors



Selecting λ with Cross-validation

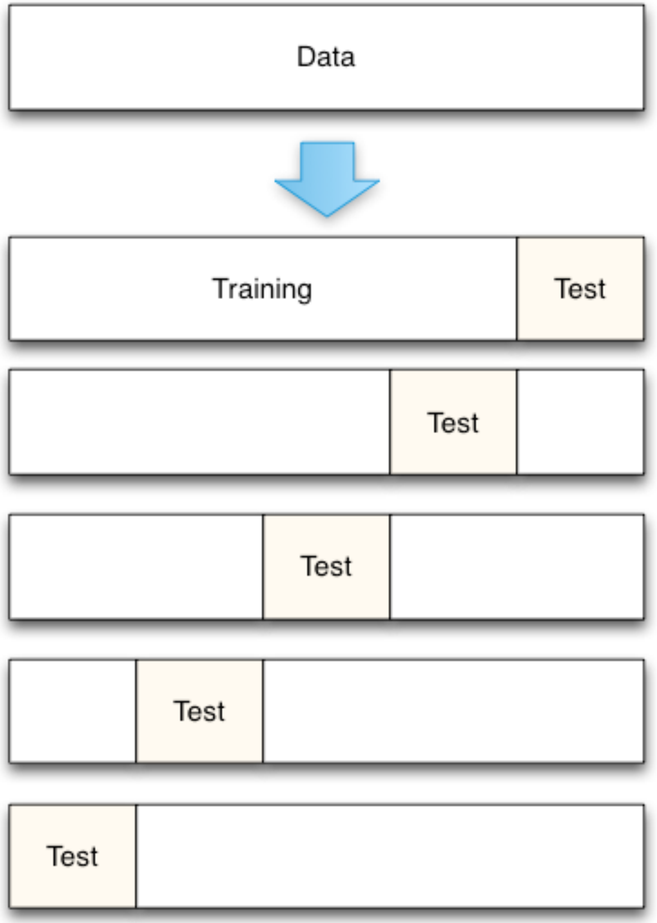
- Cross validation technique
  - Exclude part of the training data from parameter estimation
  - Use them only to predict the test error
- K-fold cross validation:
  - K splits, average K errors
- Use cross-validation for different values of λ
  - pick value that minimizes cross-validation error



Selecting λ with Cross-validation

- Cross validation technique
  - Exclude part of the training data from parameter estimation
  - Use them only to predict the test error
- K-fold cross validation:
  - K splits, average K errors
- Use cross-validation for different values of λ
  - pick value that minimizes cross-validation error

Least glorious, most effective of all methods.



Regression

1. Least Squares fitting

2. Nonlinear error function and gradient descent

3. Perceptron training (simple neural network)

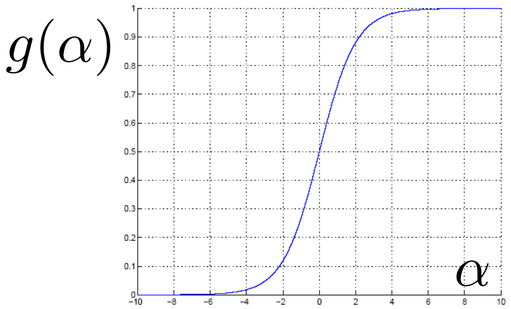
Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

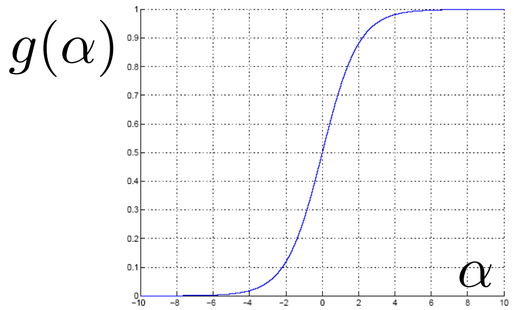
$$g(\alpha) = \frac{1}{1 + \exp(-\alpha)}$$

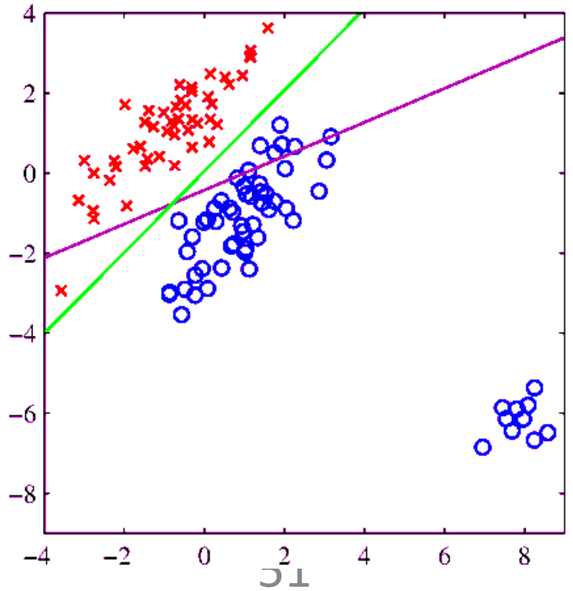


Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

$$g(\alpha) = \frac{1}{1 + \exp(-\alpha)}$$

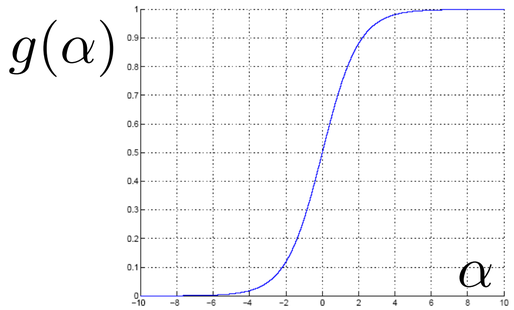


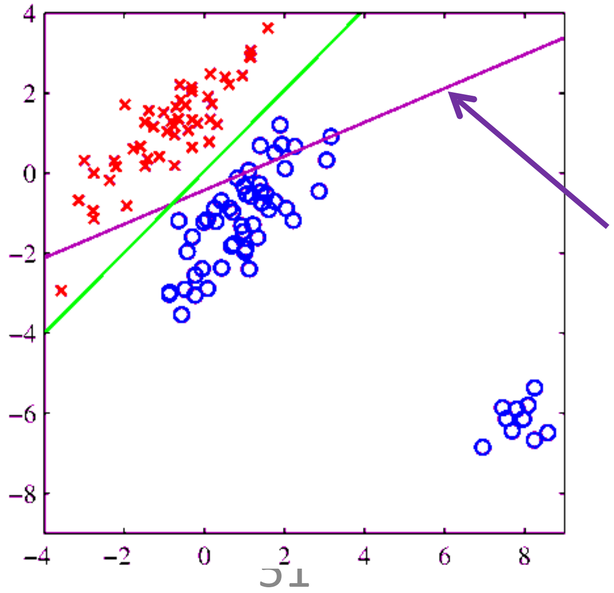


Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

$$g(\alpha) = \frac{1}{1 + \exp(-\alpha)}$$



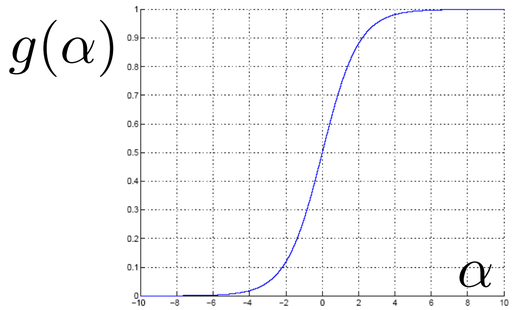


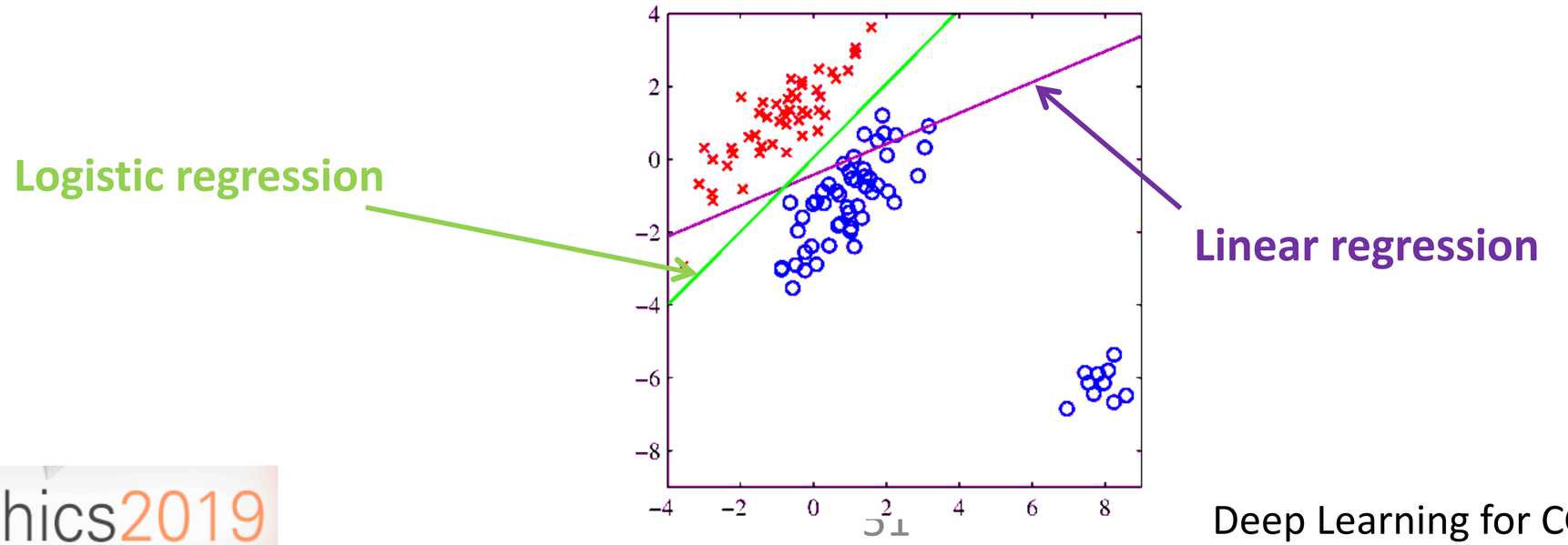
Linear regression

Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

$$g(\alpha) = \frac{1}{1 + \exp(-\alpha)}$$





Extension #2: Handling Multiple (2+) Classes

C classes: one-of-c coding (or **one-hot encoding**)

4 classes, i-th sample is in 3rd class:

$$\mathbf{y}^i = (0, 0, 1, 0)$$

Extension #2: Handling Multiple (2+) Classes

C classes: one-of-c coding (or **one-hot encoding**)

4 classes, i-th sample is in 3rd class:

$$\mathbf{y}^i = (0, 0, 1, 0)$$

Matrix notation:

$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}^1 \\ \hline \vdots \\ \hline \mathbf{y}^N \end{bmatrix} = \left[\begin{array}{c|c|c} \mathbf{y}_1 & \dots & \mathbf{y}_C \end{array} \right] \qquad \text{where} \quad \mathbf{y}_c = \begin{bmatrix} y_c^1 \\ \vdots \\ y_c^N \end{bmatrix}$$

$$\mathbf{W} = \left[\begin{array}{c|c|c} \mathbf{w}_1 & \dots & \mathbf{w}_C \end{array} \right]$$

Loss function:

$$L(\mathbf{W}) = \sum_{c=1}^{C} (\mathbf{y}_c - \mathbf{X}\mathbf{w}_c)^T (\mathbf{y}_c - \mathbf{X}\mathbf{w}_c)$$

Extension #2: Handling Multiple (2+) Classes

C classes: one-of-c coding (or **one-hot encoding**)

4 classes, i-th sample is in 3rd class:
$\mathbf{y}^i = (0, 0, 1, 0)$

Matrix notation:

$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}^1 \\ \hline \vdots \\ \hline \mathbf{y}^N \end{bmatrix} = \left[\begin{array}{c|c|c} \mathbf{y}_1 & \dots & \mathbf{y}_C \end{array} \right] \quad \text{where} \quad \mathbf{y}_c = \begin{bmatrix} y_c^1 \\ \vdots \\ y_c^N \end{bmatrix}$$

$$\mathbf{W} = \left[\begin{array}{c|c|c} \mathbf{w}_1 & \dots & \mathbf{w}_C \end{array} \right]$$

Loss function:

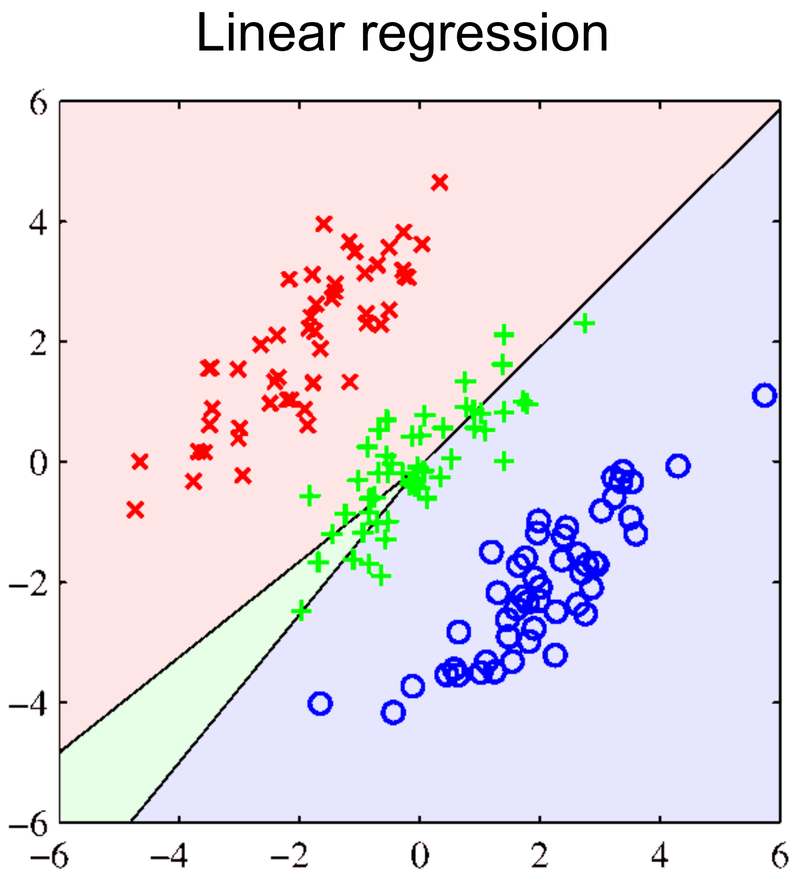
$$L(\mathbf{W}) = \sum_{c=1}^{C} (\mathbf{y}_c - \mathbf{X}\mathbf{w}_c)^T (\mathbf{y}_c - \mathbf{X}\mathbf{w}_c)$$

Least squares fit (decouples per class):

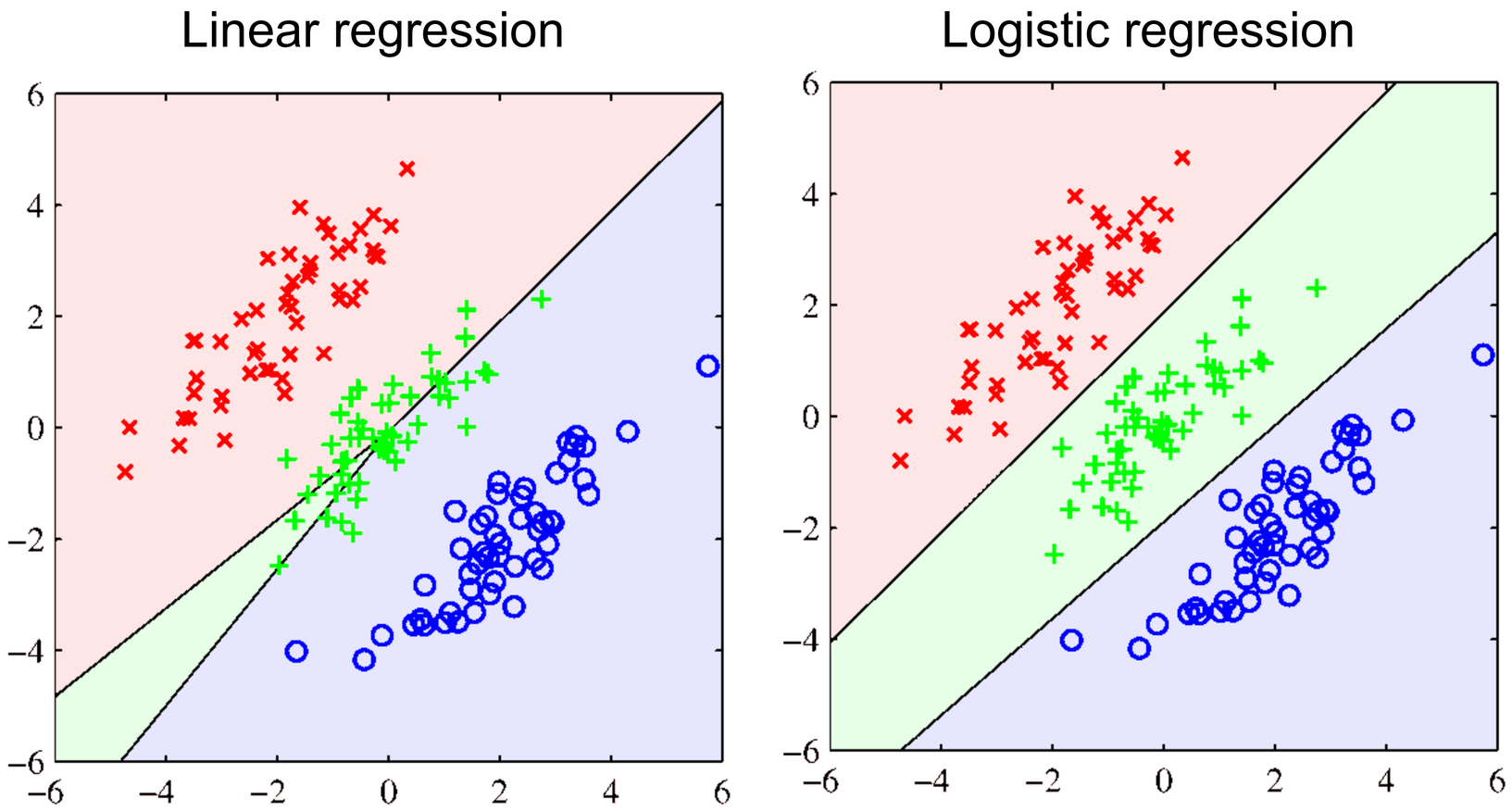
$$\mathbf{w}_c^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}_c$$

Logistic vs Linear Regression, n>2 classes

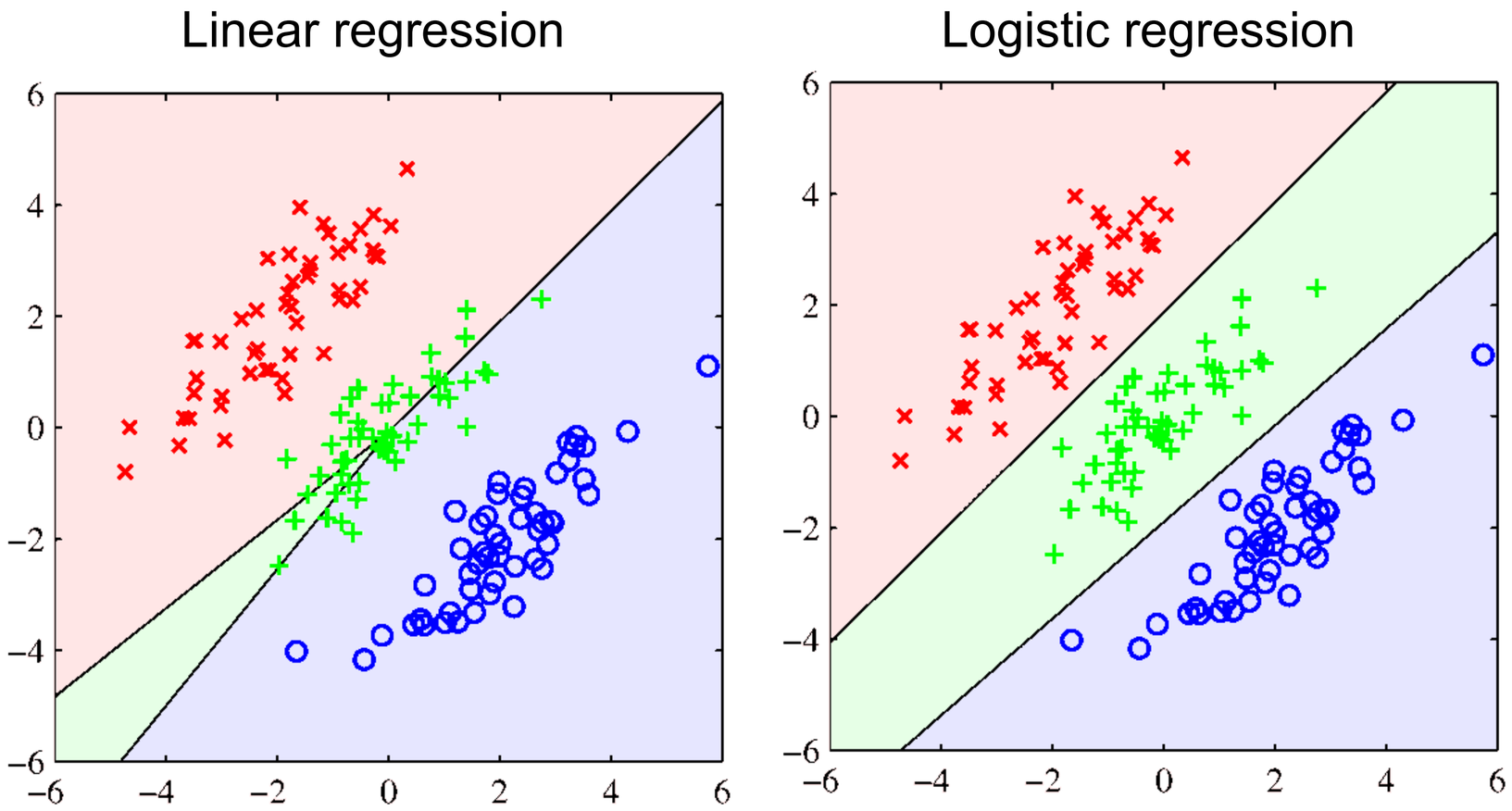
Logistic vs Linear Regression, n>2 classes



Logistic vs Linear Regression, n>2 classes



Logistic vs Linear Regression, n>2 classes



Logistic regression does not exhibit the masking problem

Gradient of Cross-entropy Loss

$$L(\mathbf{w}) = -\sum_{i=1}^{N} y^i \log g(\mathbf{w}^T \mathbf{x}^i) + (1 - y^i) \log(1 - g(\mathbf{w}^T \mathbf{x}^i))$$

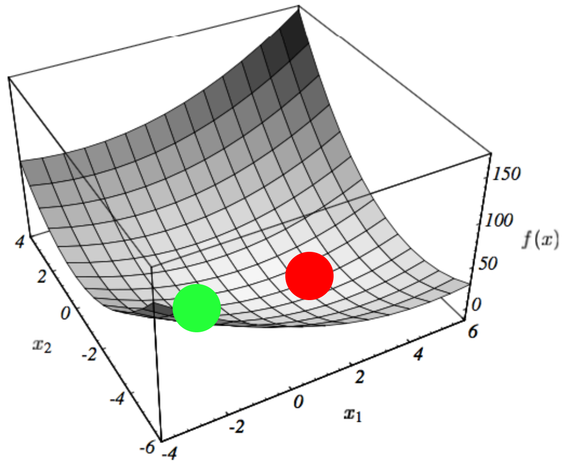
Gradient of Cross-entropy Loss

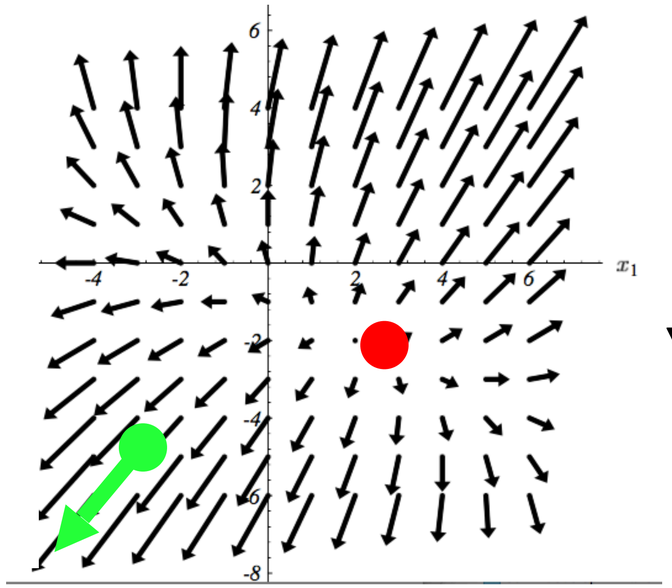
$$L(\mathbf{w}) = -\sum_{i=1}^{N} y^i \log g(\mathbf{w}^T \mathbf{x}^i) + (1 - y^i) \log(1 - g(\mathbf{w}^T \mathbf{x}^i))$$

$$\nabla L(\mathbf{w}^*) = \mathbf{0}$$

nonlinear system of equations!!

Gradient Descent Minimization

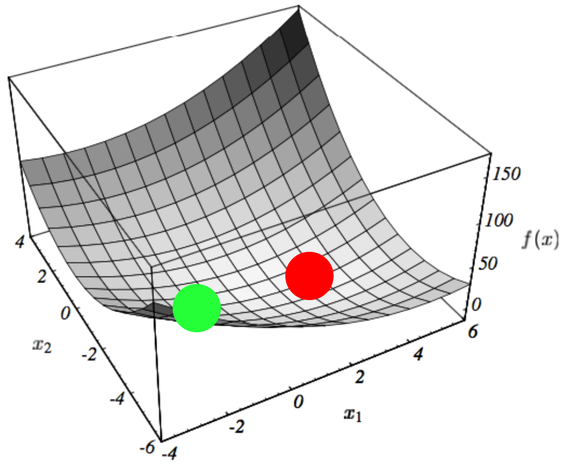


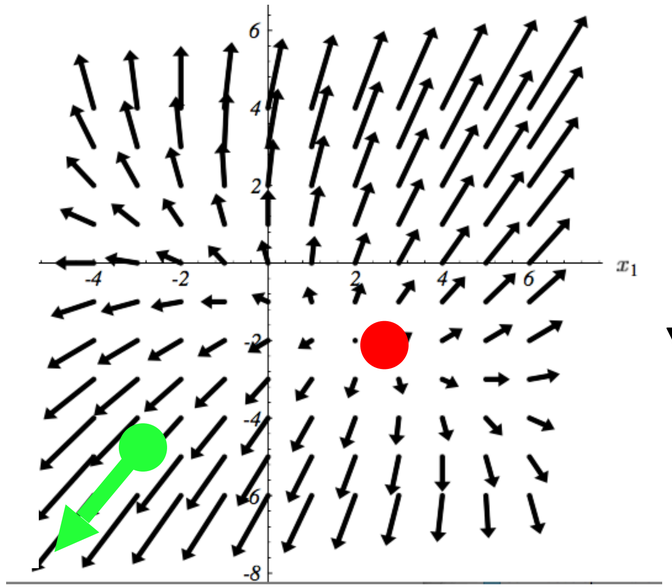


$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

gradient at any point gives direction of fastest increase

Gradient Descent Minimization



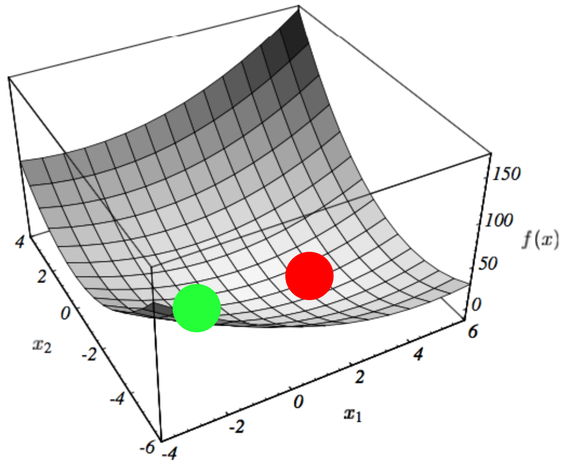


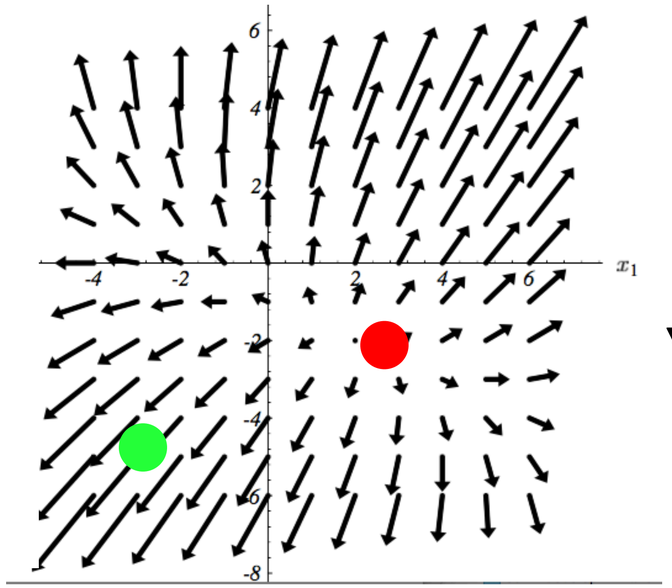
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

gradient at any point gives direction of fastest increase

Idea: start at a point and move in the direction opposite to the gradient

Gradient Descent Minimization



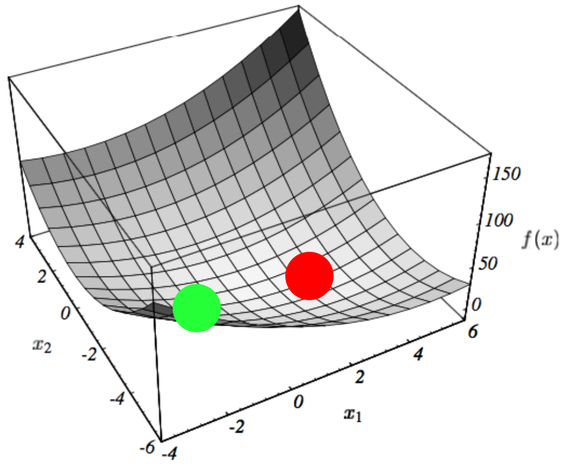


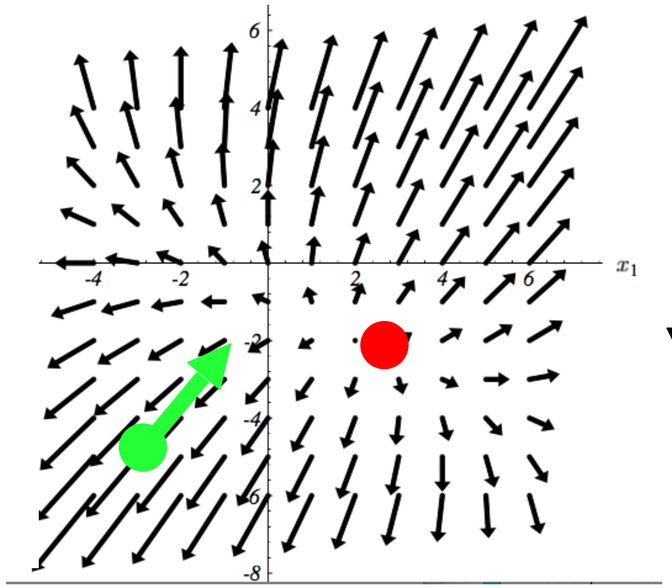
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

gradient at any point gives direction of fastest increase

Idea: start at a point and move in the direction opposite to the gradient

Gradient Descent Minimization



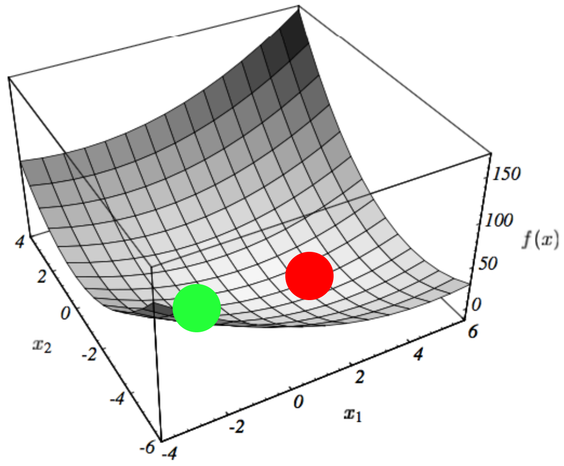


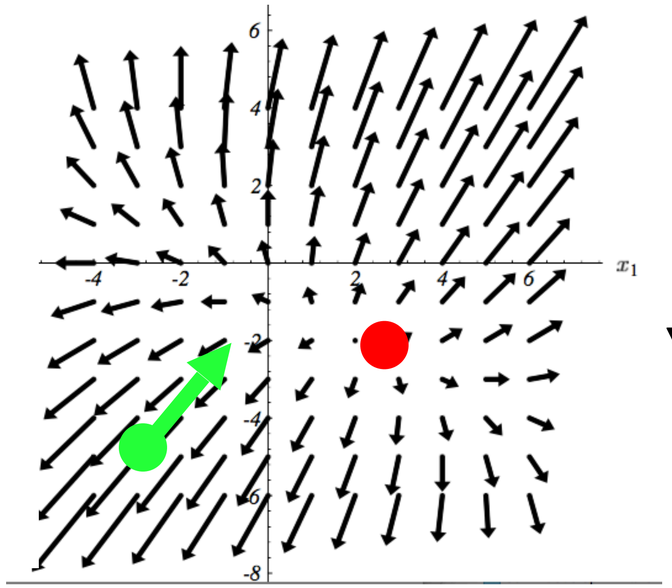
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

gradient at any point gives direction of fastest increase

Idea: start at a point and move in the direction opposite to the gradient

Gradient Descent Minimization





$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

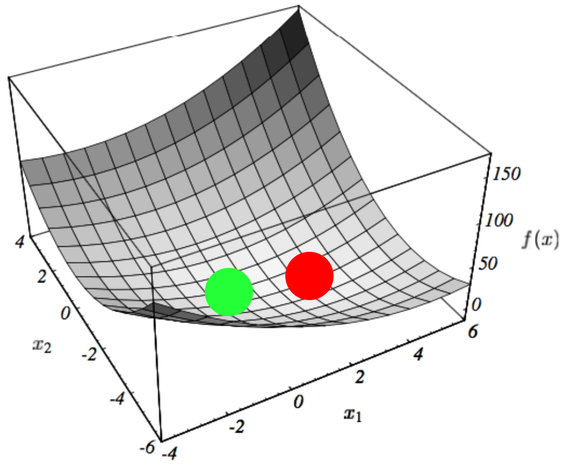
gradient at any point gives direction of fastest increase

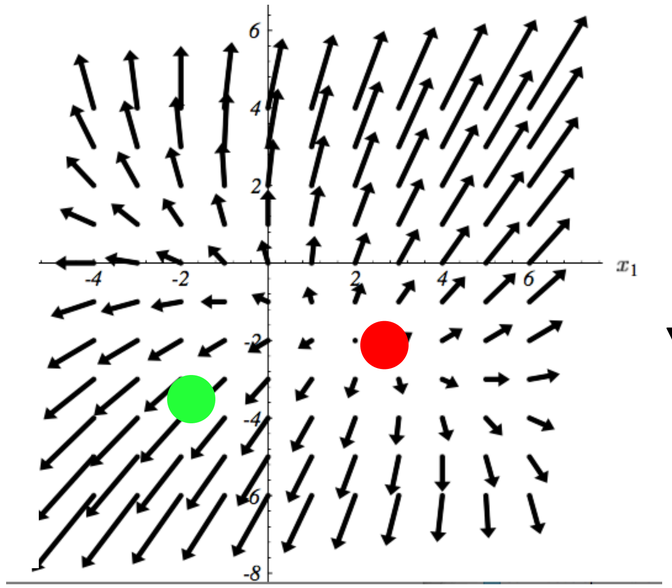
Idea: start at a point and move in the direction opposite to the gradient

Initialize: $\mathbf{x}_0$

Update: $\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$ **i=0**

Gradient Descent Minimization

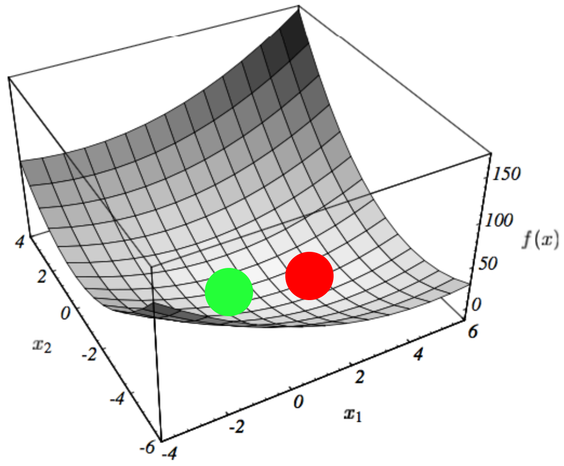


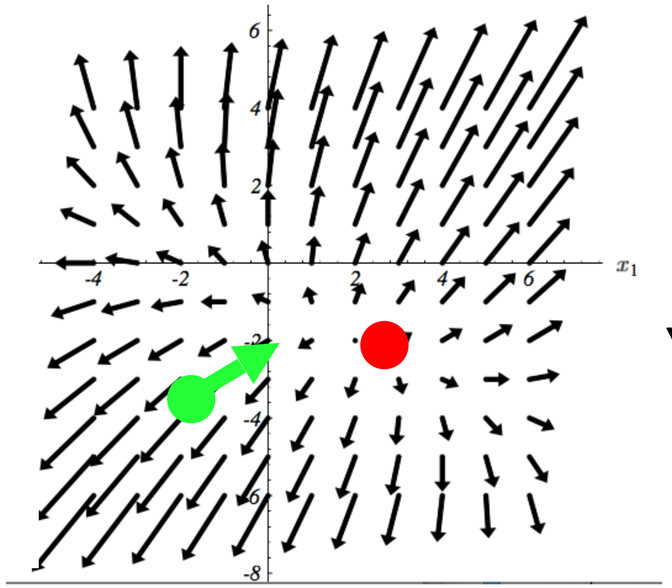


$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

Update: $$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$$ **i=1**

Gradient Descent Minimization

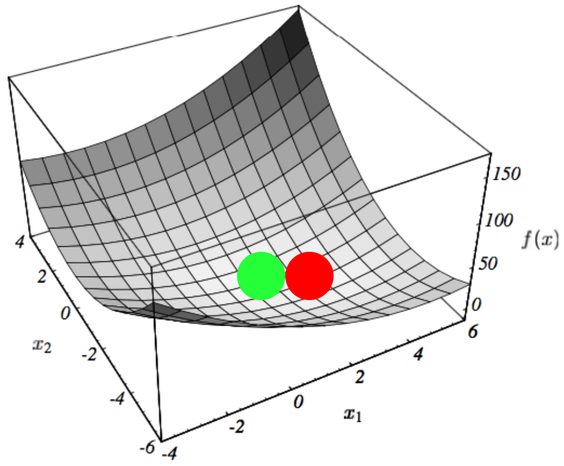


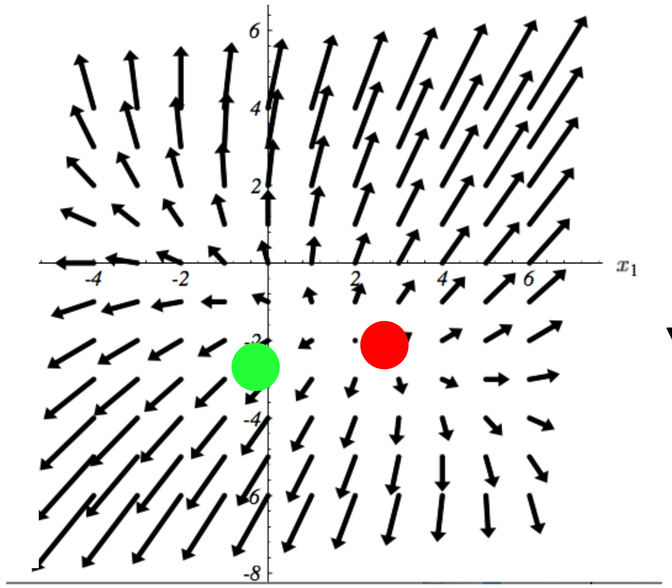


$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

Update: $$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$$ **i=1**

Gradient Descent Minimization

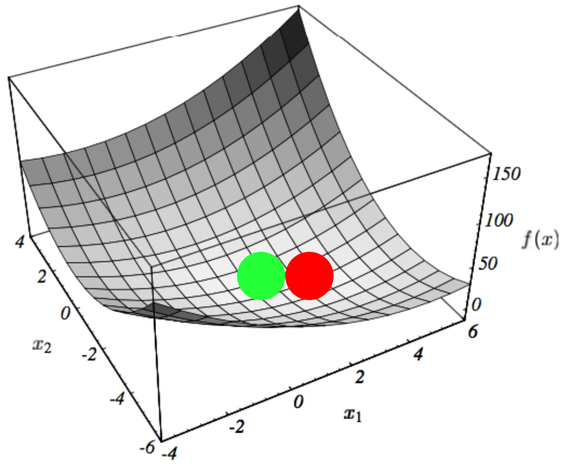


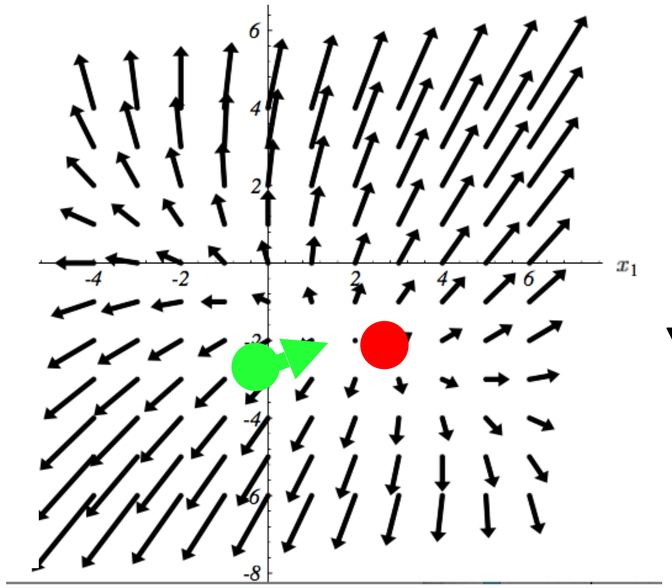


$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

Update: $$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$$ **i=2**

Gradient Descent Minimization

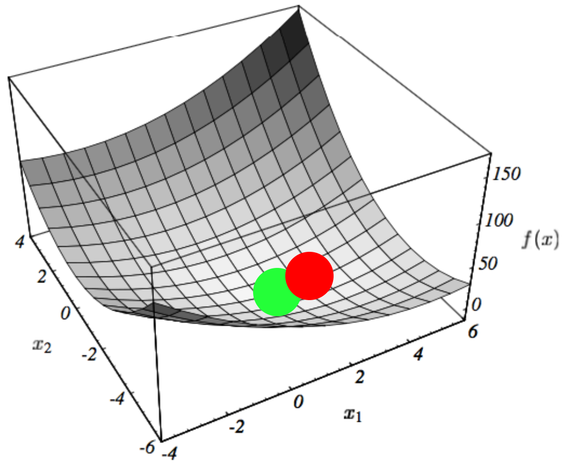


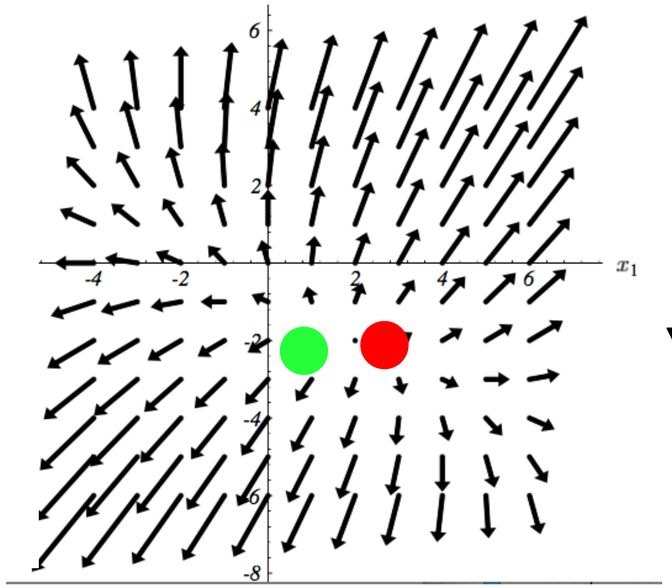


$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

Update: $\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$ **i=2**

Gradient Descent Minimization





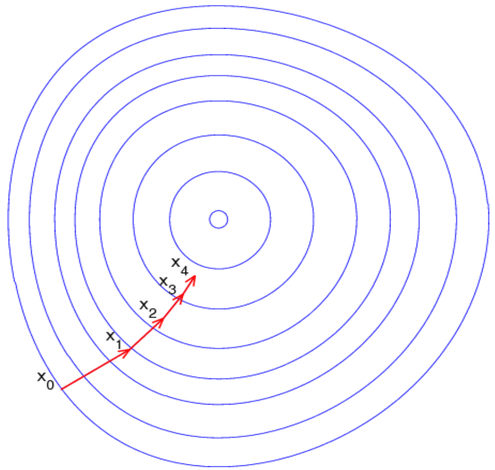
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

Update: $$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$$ **i=3**

Gradient Descent Minimization

Initialize: $\mathbf{x}_0$

Update: $\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$

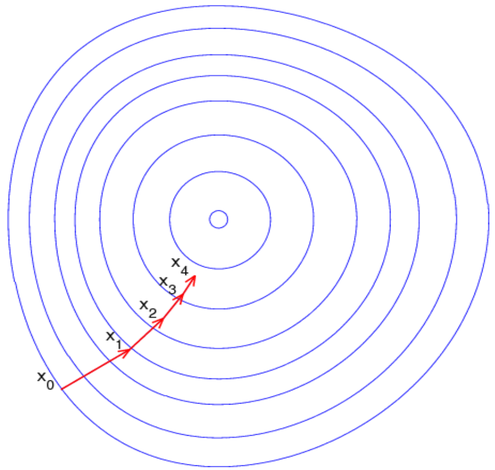


Gradient Descent Minimization

Initialize: $\mathbf{x}_0$

Update: $\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$

We can always make it converge for a convex function.

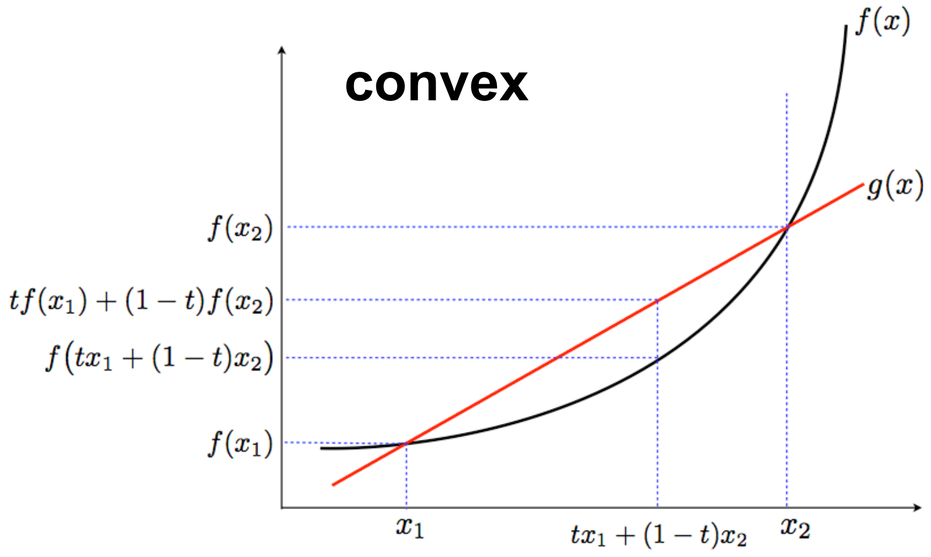


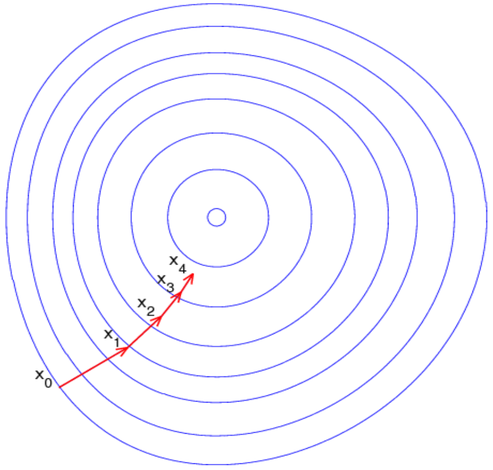
Gradient Descent Minimization

Initialize: $\mathbf{x}_0$

Update: $\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$

We can always make it converge for a convex function.



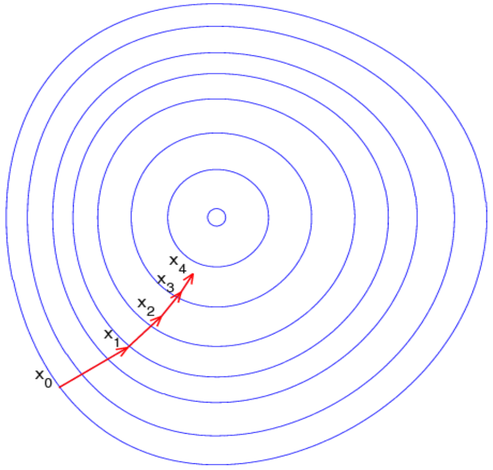


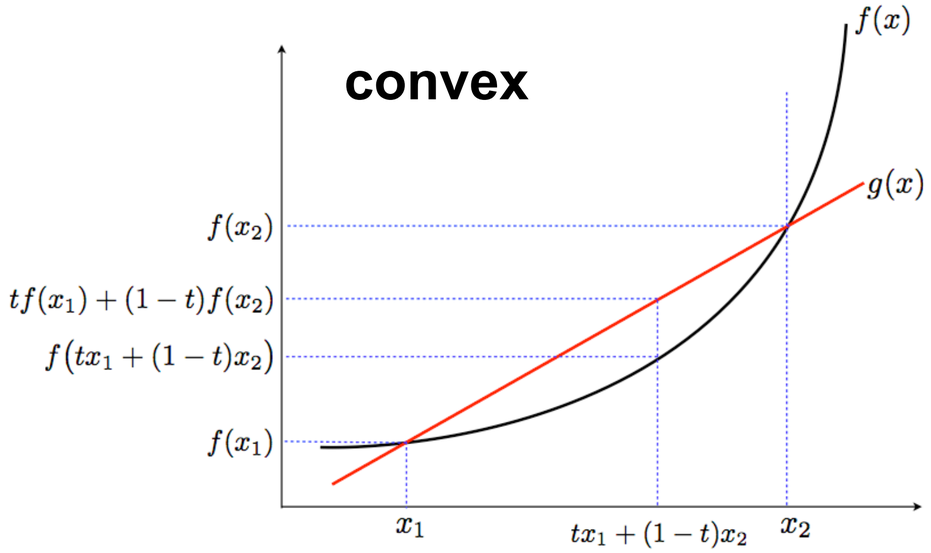
Gradient Descent Minimization

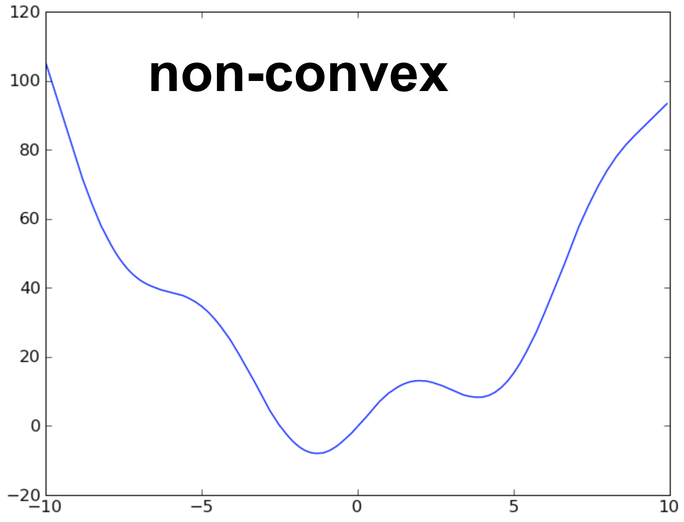
Initialize: $\mathbf{x}_0$

Update: $\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$

We can always make it converge for a convex function.







XOR Problem

XOR Problem

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

XOR Problem

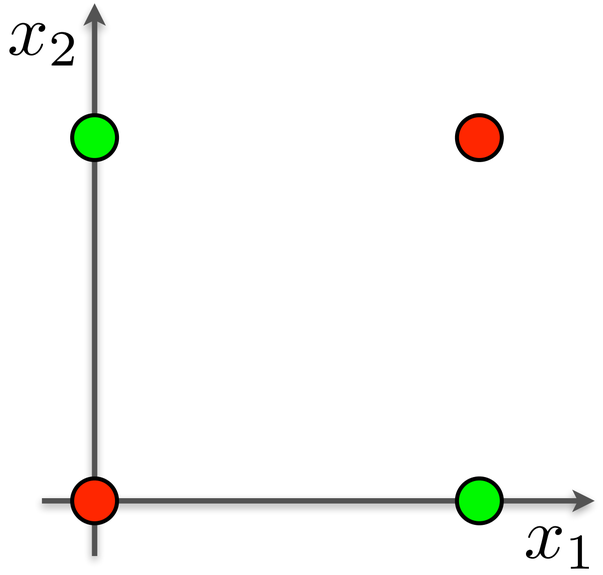
$$y = f(x_1, x_2)$$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

XOR Problem

$$y = f(x_1, x_2)$$

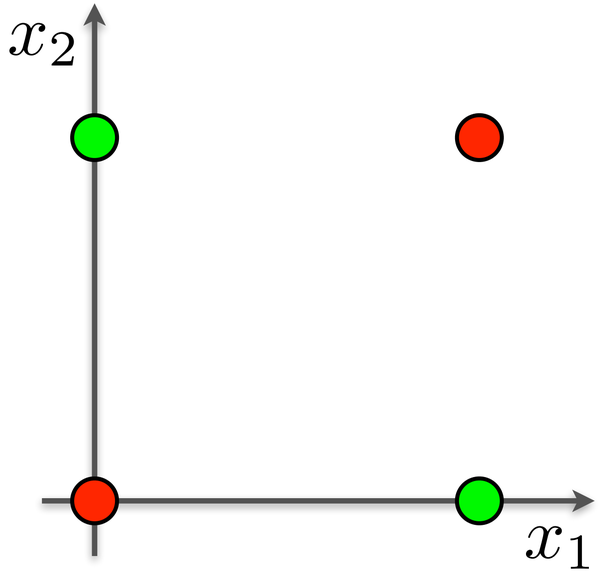
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



XOR Problem

$$y = f(x_1, x_2)$$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

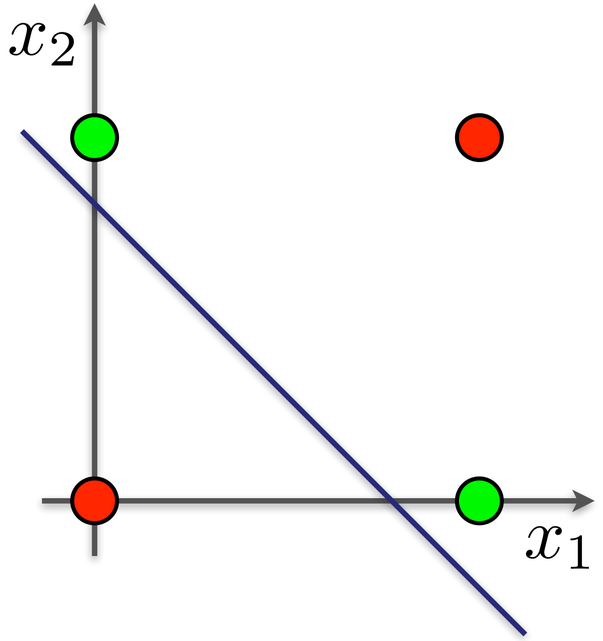


$$y = f(w_0, w_1, w_2) = \mathcal{H}(w_0 + w_1 x_1 + w_2 x_2)$$

XOR Problem

$$y = f(x_1, x_2)$$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

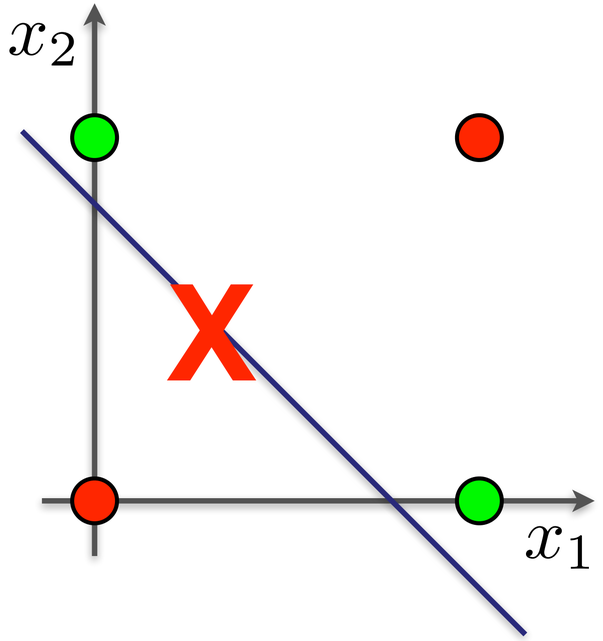


$$y = f(w_0, w_1, w_2) = \mathcal{H}(w_0 + w_1 x_1 + w_2 x_2)$$

XOR Problem

$$y = f(x_1, x_2)$$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



$$y = f(w_0, w_1, w_2) = \mathcal{H}(w_0 + w_1 x_1 + w_2 x_2)$$

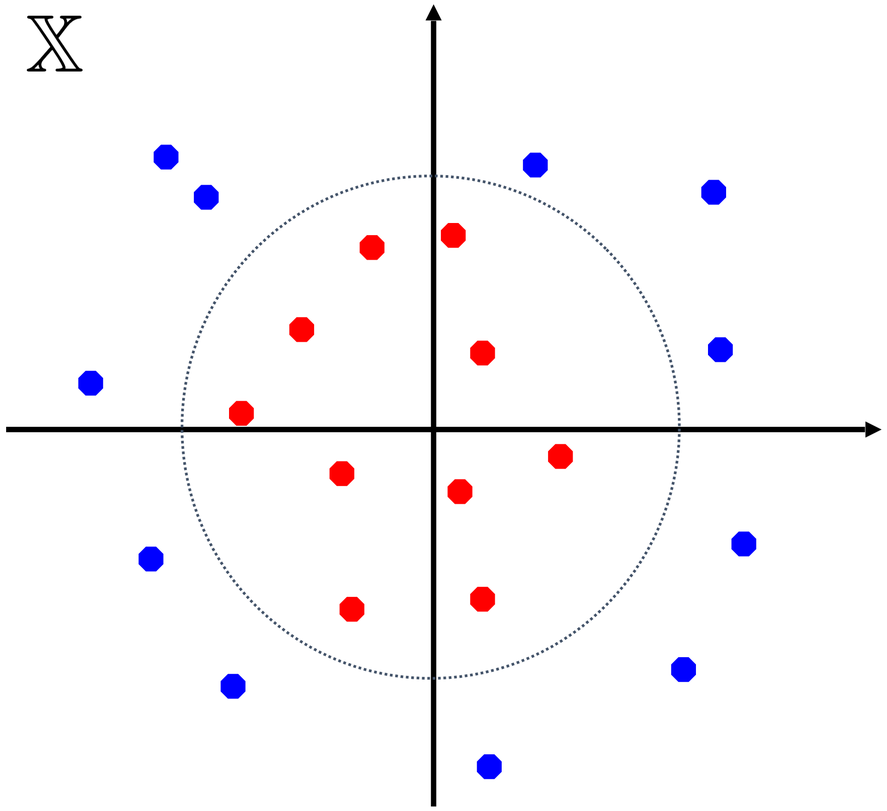
Regression

1. Least Squares fitting

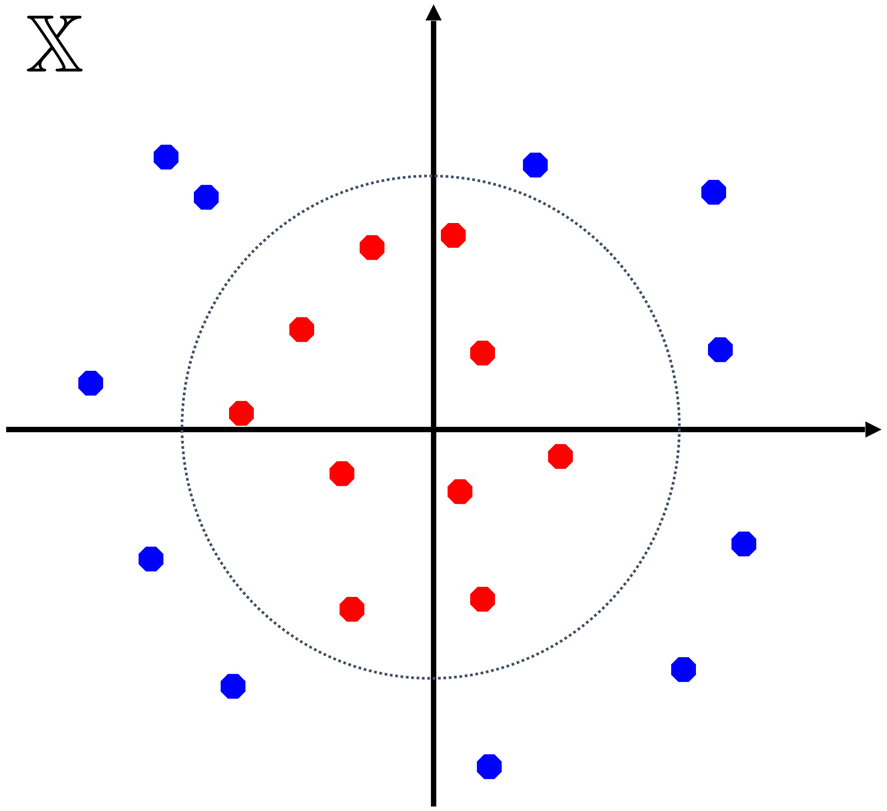
2. Nonlinear error function and gradient descent

3. Perceptron training (simple neural network)

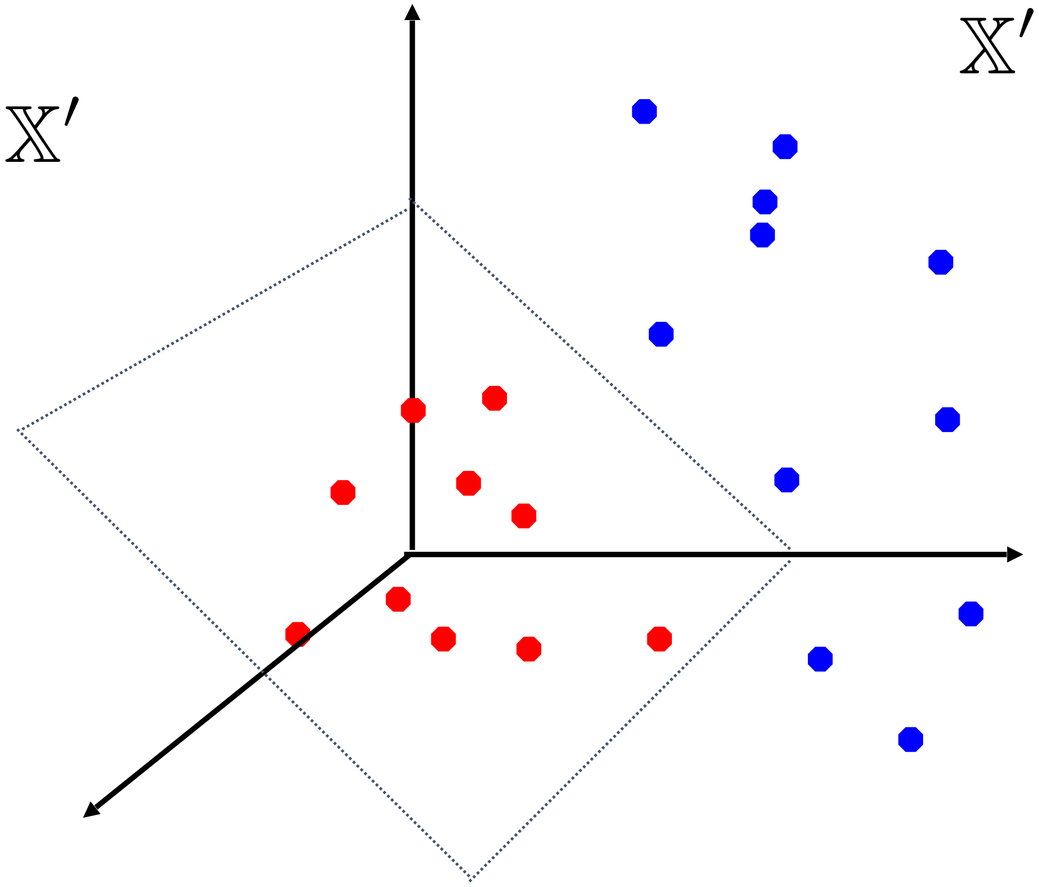
Lifting to Higher Dimensions



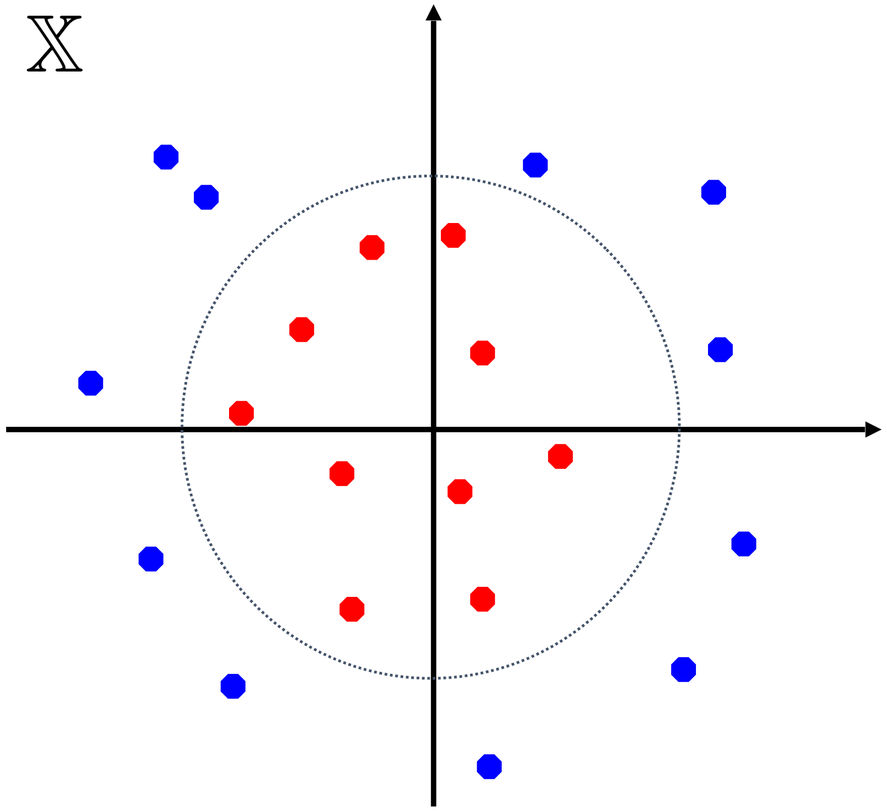
Lifting to Higher Dimensions



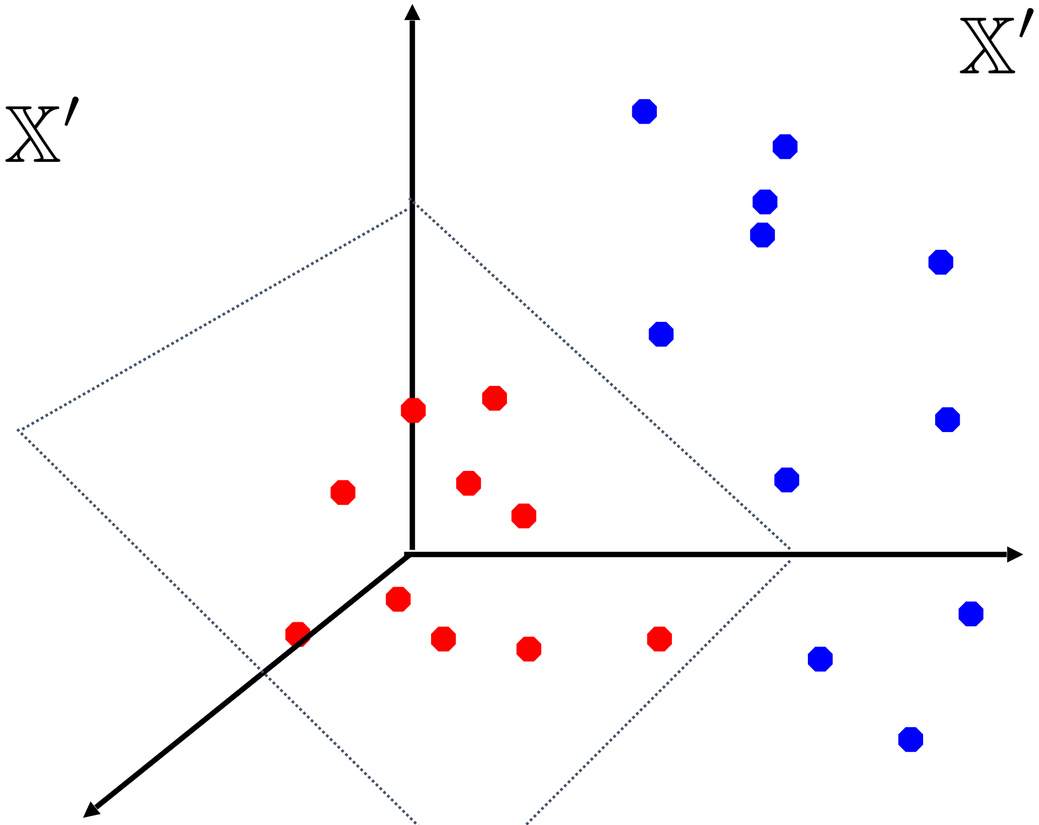
$$g : \mathbb{X} \longrightarrow \mathbb{X}'$$



Lifting to Higher Dimensions



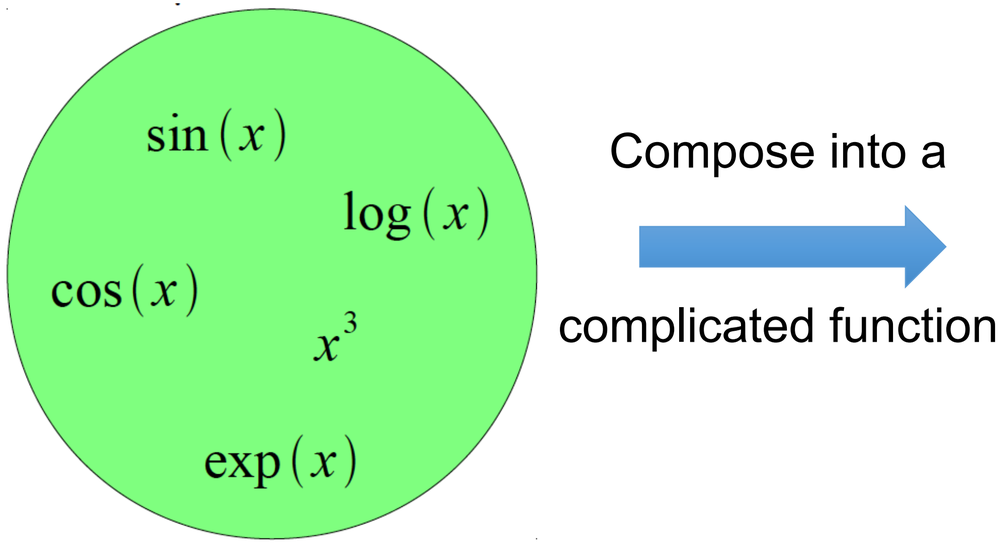
$$g : \mathbb{X} \longrightarrow \mathbb{X}'$$



$$f_\theta(x) = \begin{cases} 1 & \text{if } w\ g(x) + b \geq 0 \\ 0 & \text{if } w\ g(x) + b < 0 \end{cases}$$

Building A Complicated Function

Given a library of simple functions



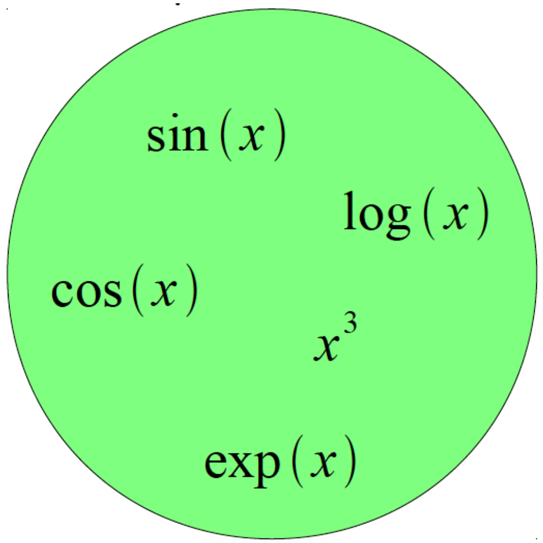
Compose into a

complicated function

Slide Credit: Marc'Aurelio Ranzato, Yann LeCun

Building A Complicated Function

Given a library of simple functions



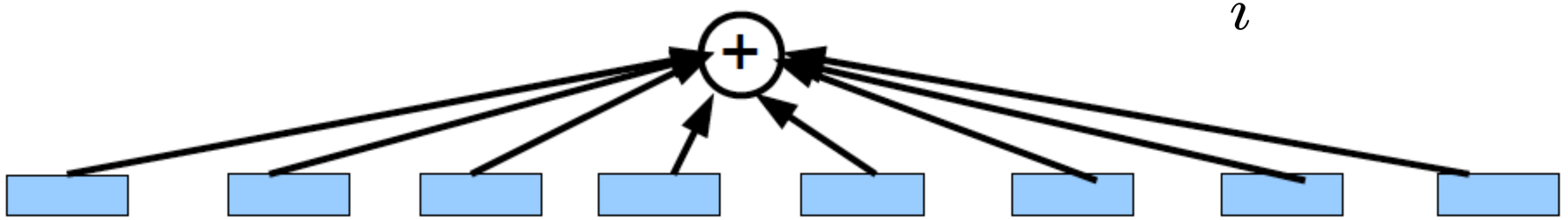
Compose into a

complicated function

Idea 1: Linear Combinations

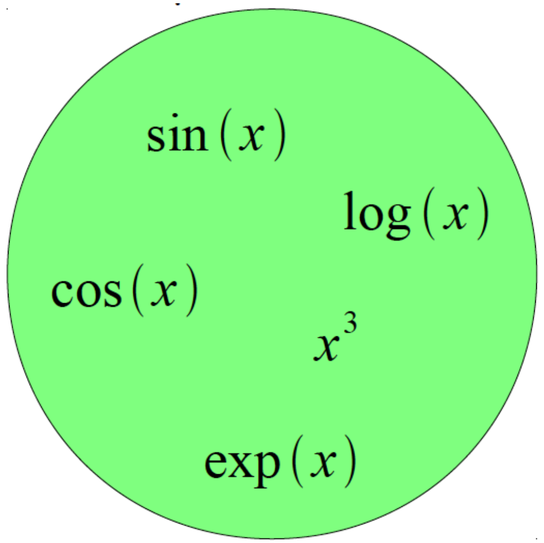
- Boosting
- Kernels
- …

$$f(x) = \sum_i \alpha_i g_i(x)$$



Building A Complicated Function

Given a library of simple functions



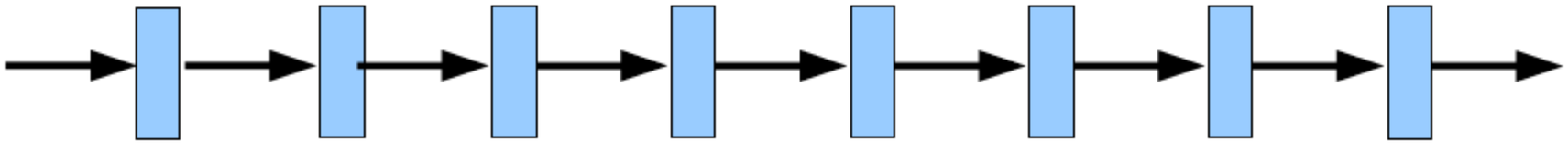
Compose into a

complicated function

Idea 2: Compositions

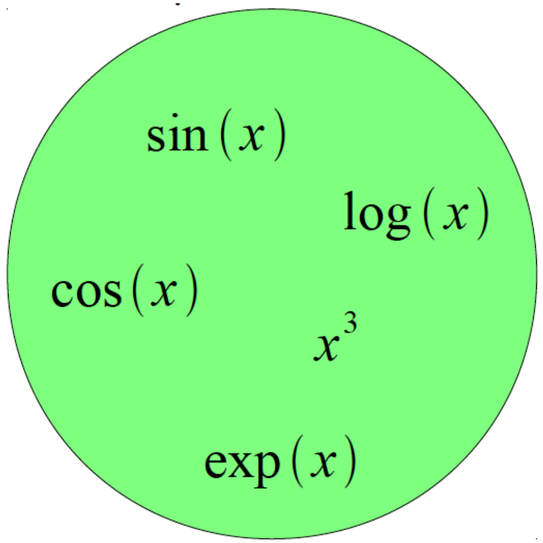
- Decision Trees
- Deep Learning

$$f(x) = g_1(g_2(\ldots(g_n(x)\ldots))$$



Building A Complicated Function

Given a library of simple functions



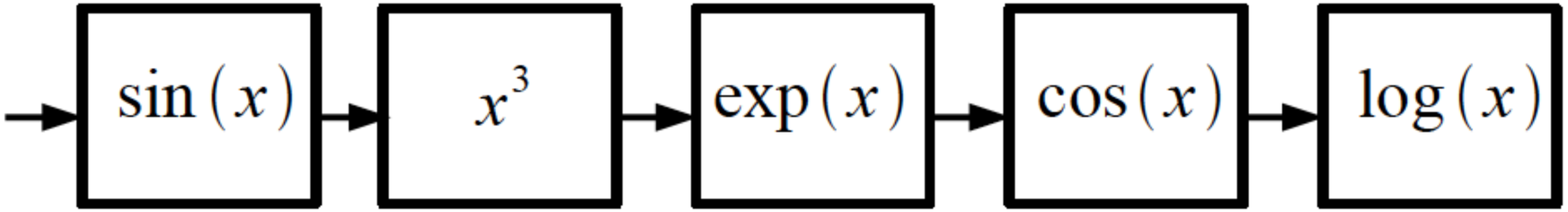
Compose into a

complicated function

Idea 2: Compositions

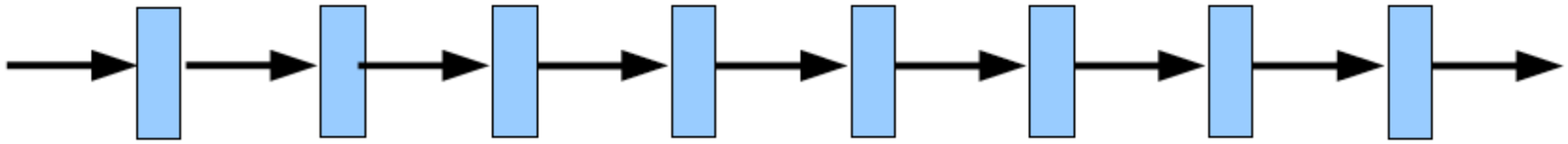
- Decision Trees
- Deep Learning

$$f(x) = \log(\cos(\exp(\sin^3(x))))$$



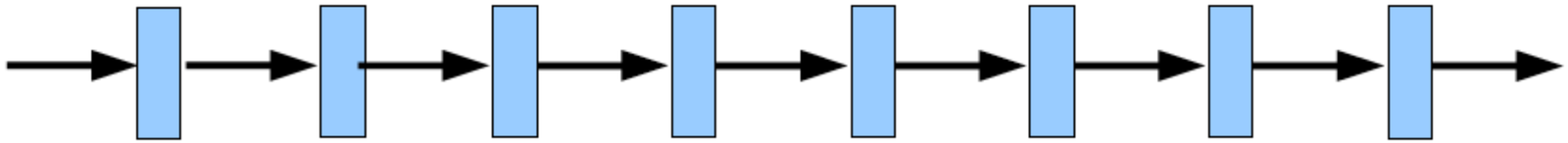
‘Neuron’: Cascade of Linear and Nonlinear Function

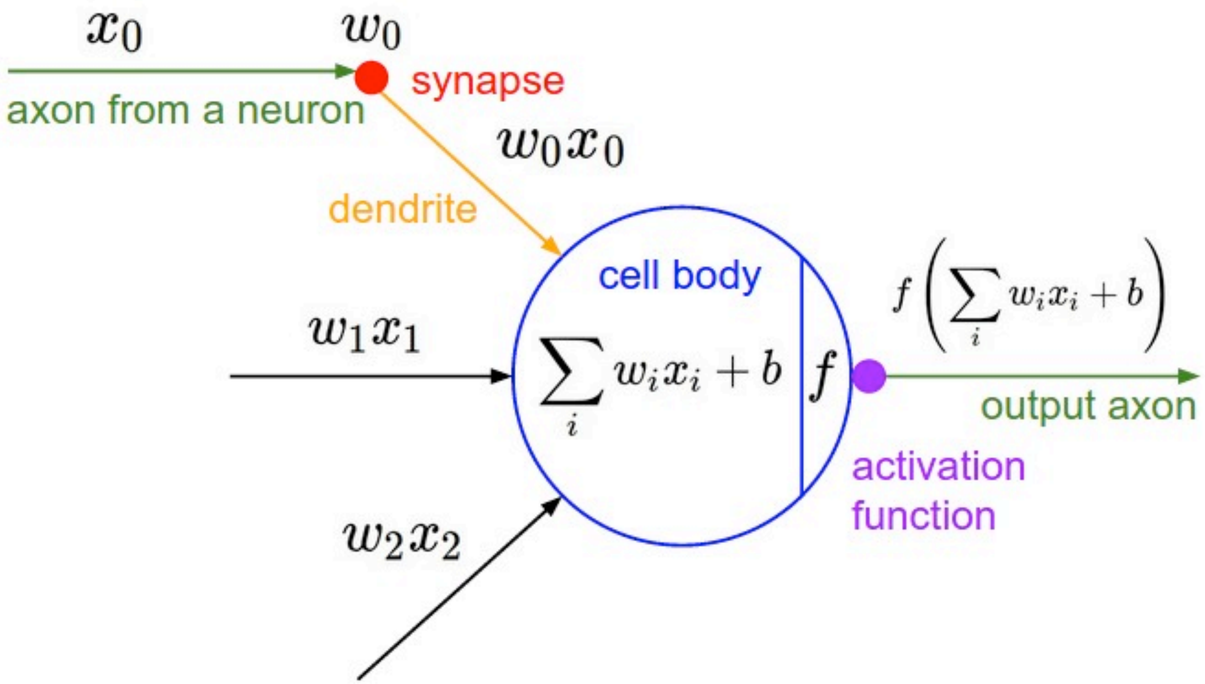
basic building block



‘Neuron’: Cascade of Linear and Nonlinear Function

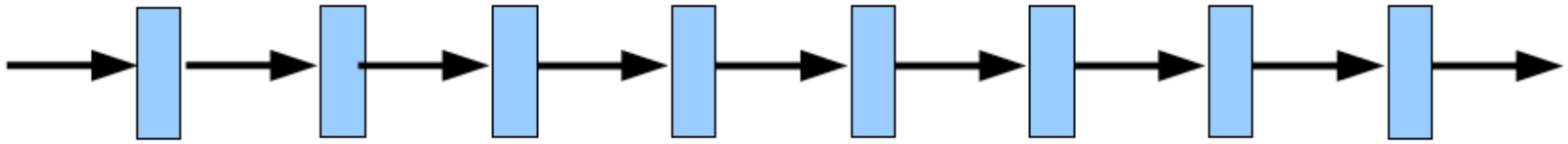
basic building block

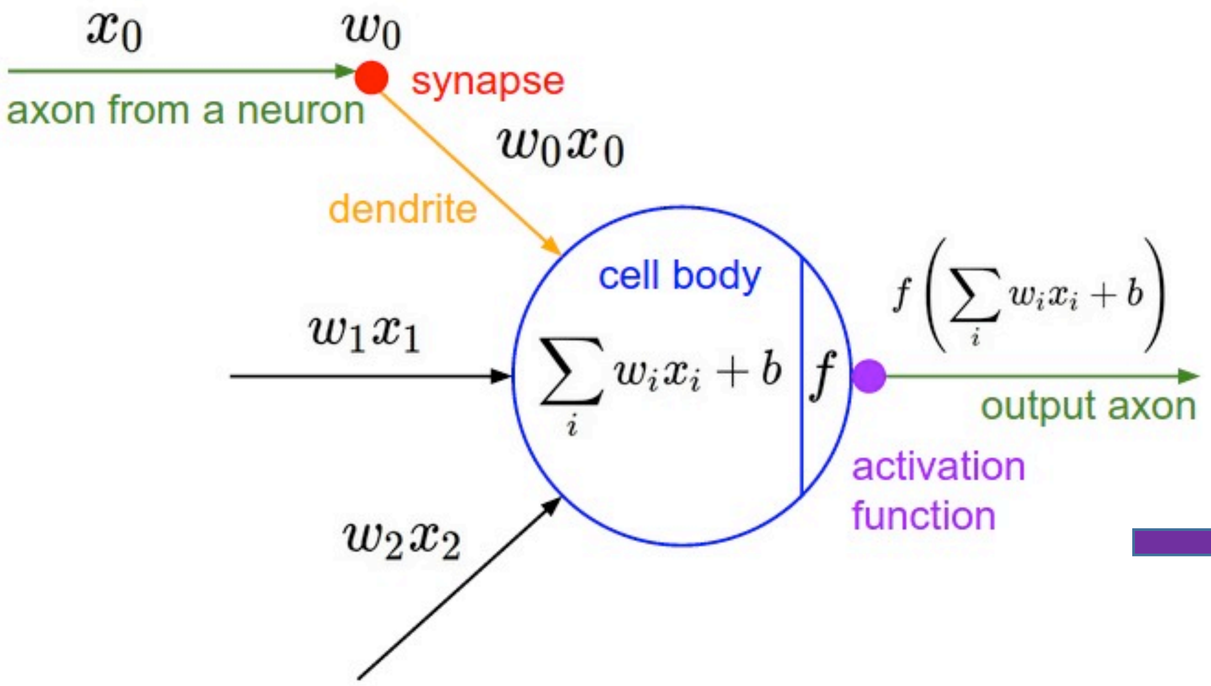




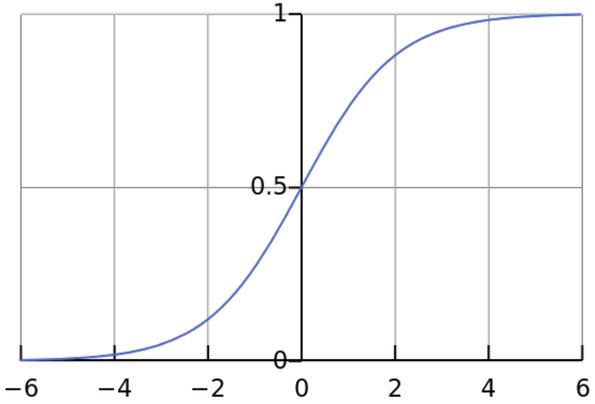
‘Neuron’: Cascade of Linear and Nonlinear Function

basic building block

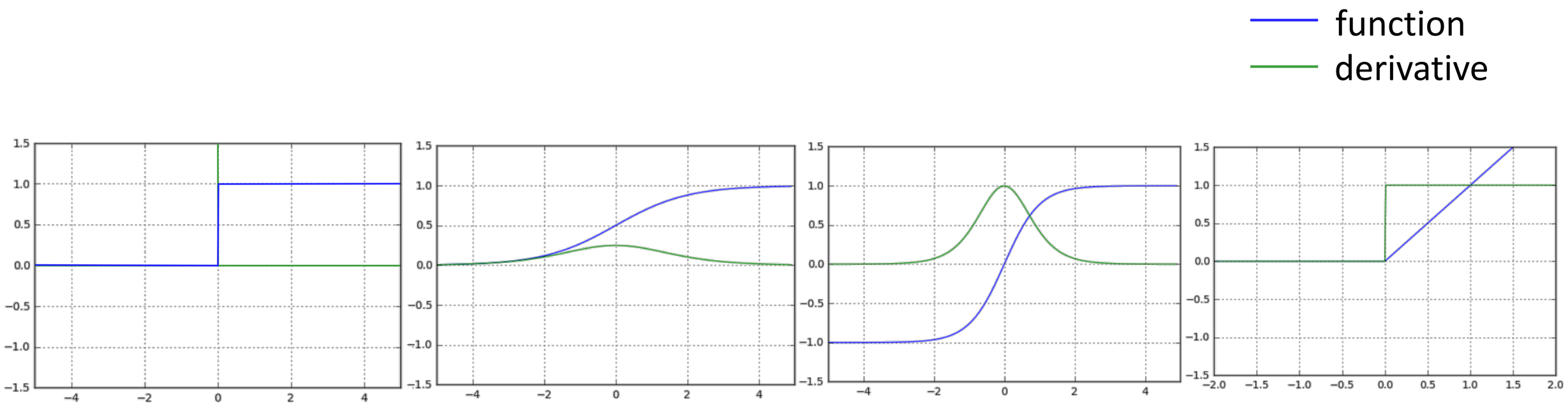




Sigmoidal activation



Activation Functions



Step ("perceptron")

$$g(a) = \begin{cases} 0 & a < 0 \\ 1 & a \geq 0 \end{cases}$$

Sigmoidal ("logistic")

$$g(a) = \frac{1}{1 + \exp(-a)}$$

Hyperbolic tangent

$$g(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$

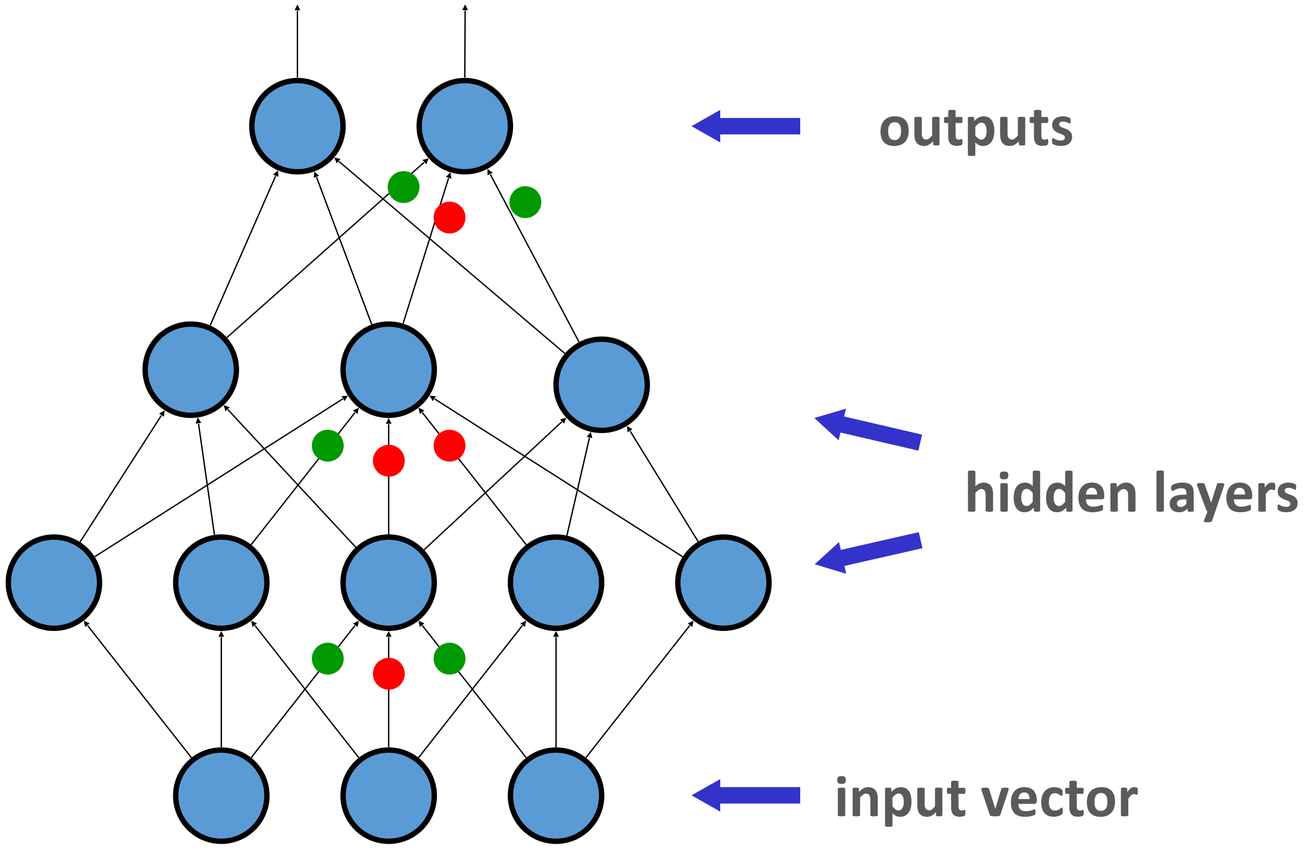
Rectified Linear Unit (RELU)

$$g(a) = \max(0, a)$$

Image Credit: Olivier Grisel and Charles Ollion

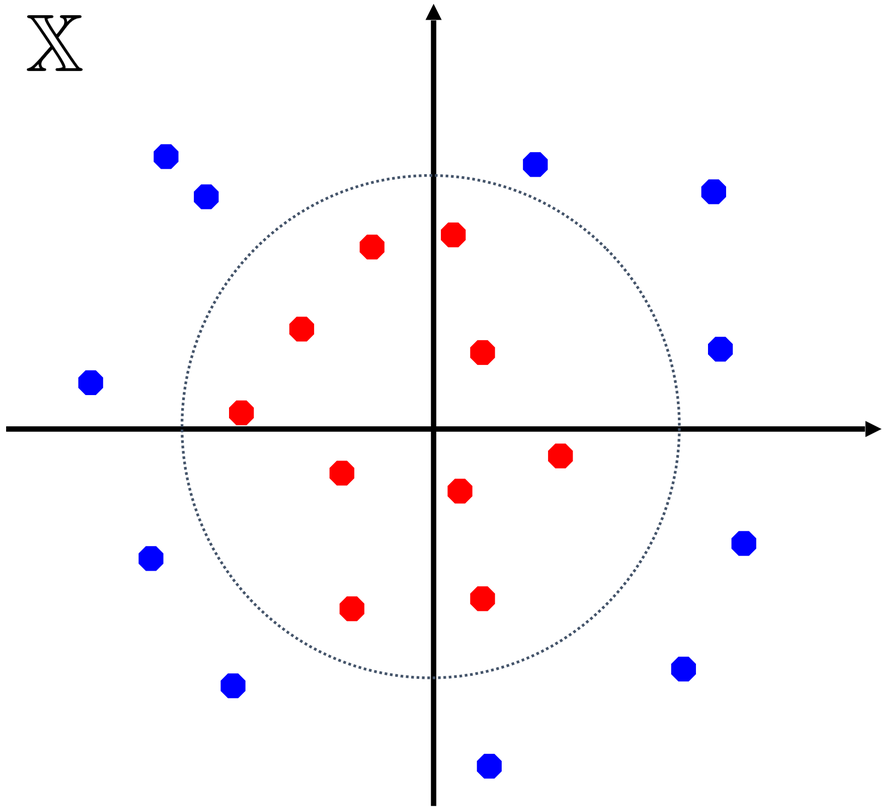
Multi-Layer Perceptrons (~1985)

$$u_i = g\left(\sum_{k \in \mathcal{N}(i)} w_{k,i}\, g\left(\sum_{m \in \mathcal{N}(k)} w_{m,k} u_m + b_k\right) + b_i\right)$$

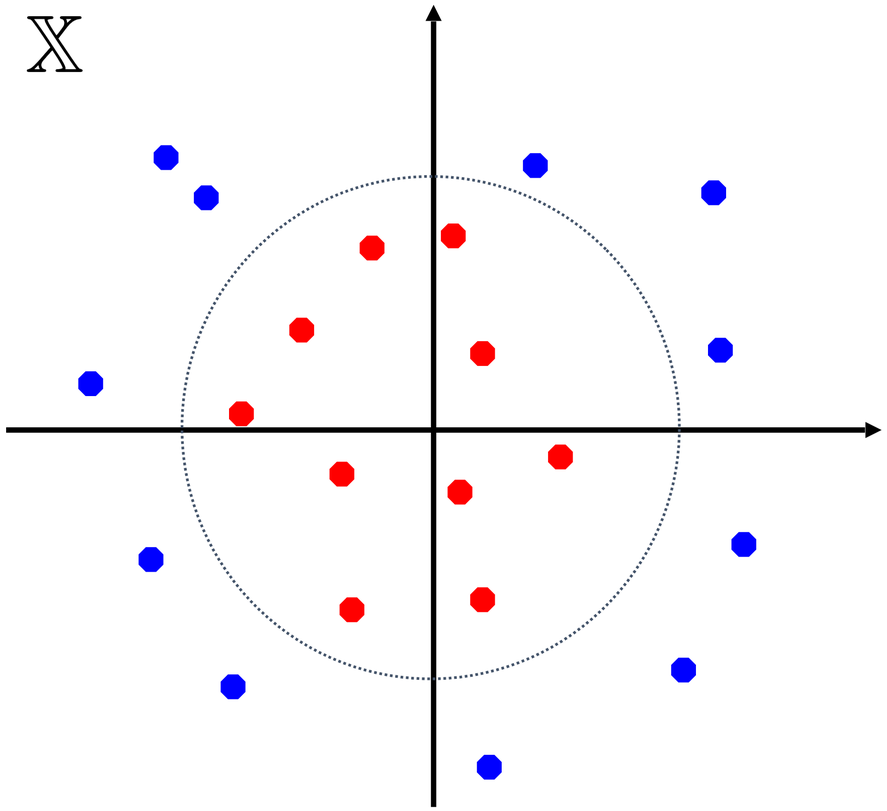


Slide credit: G. Hinton

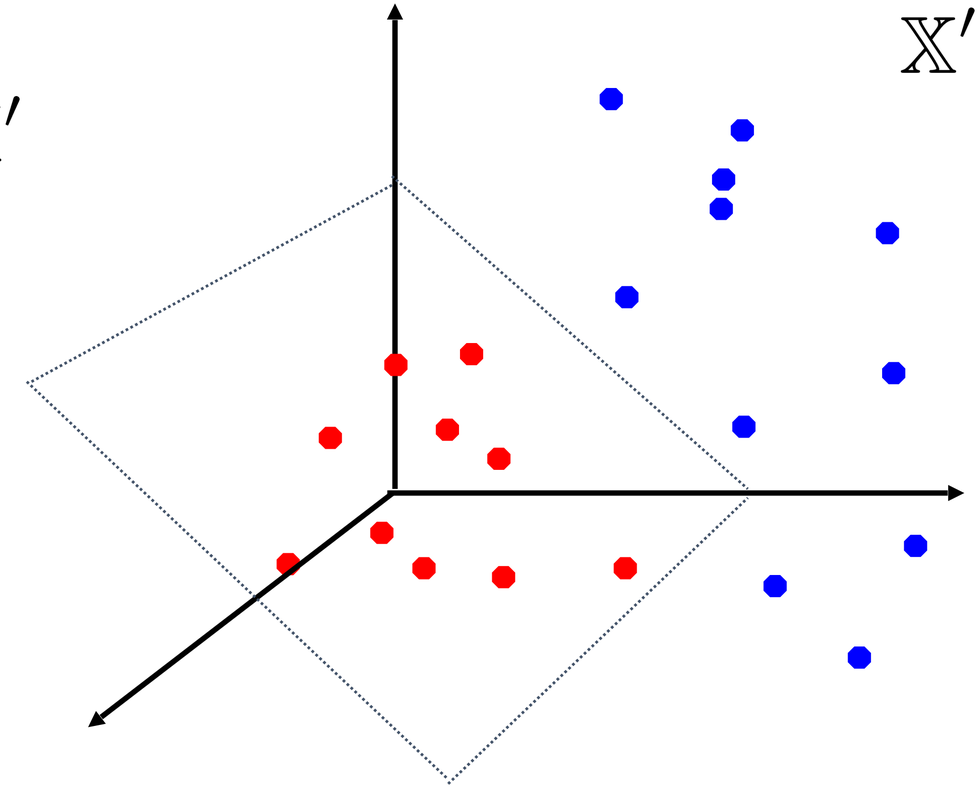
Reminder: Non-linear Decision Boundaries



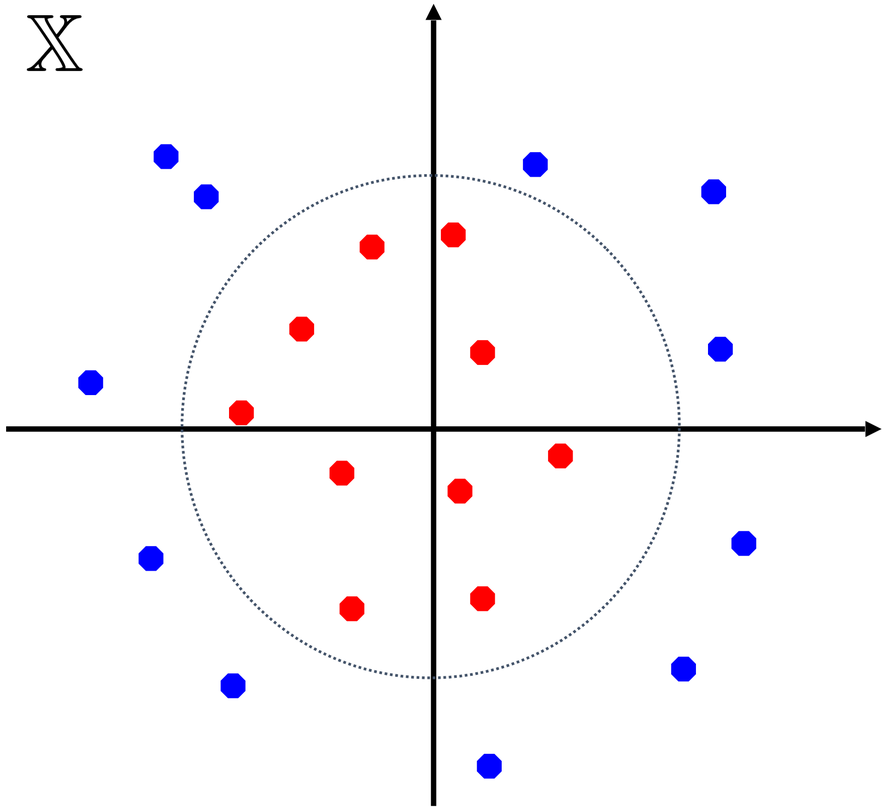
Reminder: Non-linear Decision Boundaries



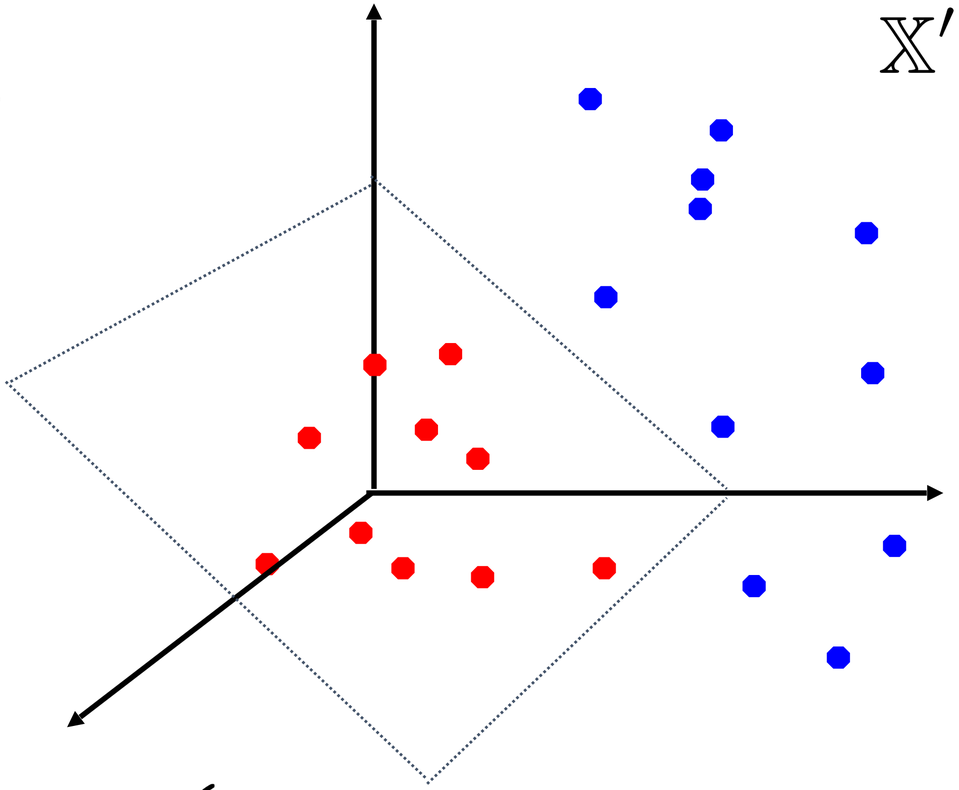
$$g : \mathbb{X} \longrightarrow \mathbb{X}'$$



Reminder: Non-linear Decision Boundaries



$$g : \mathbb{X} \longrightarrow \mathbb{X}'$$

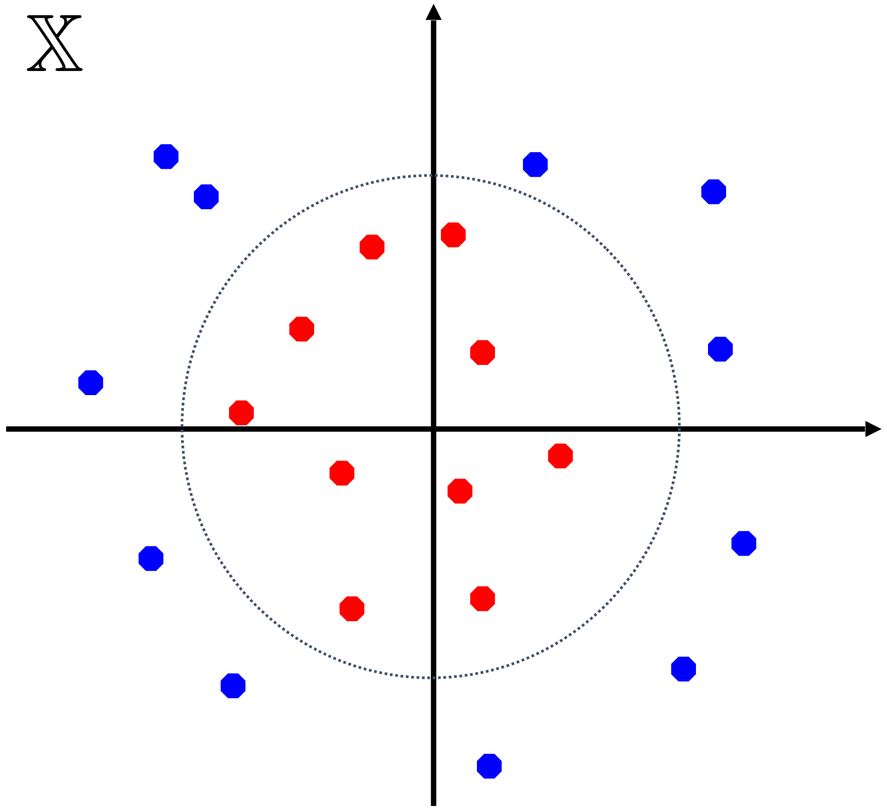


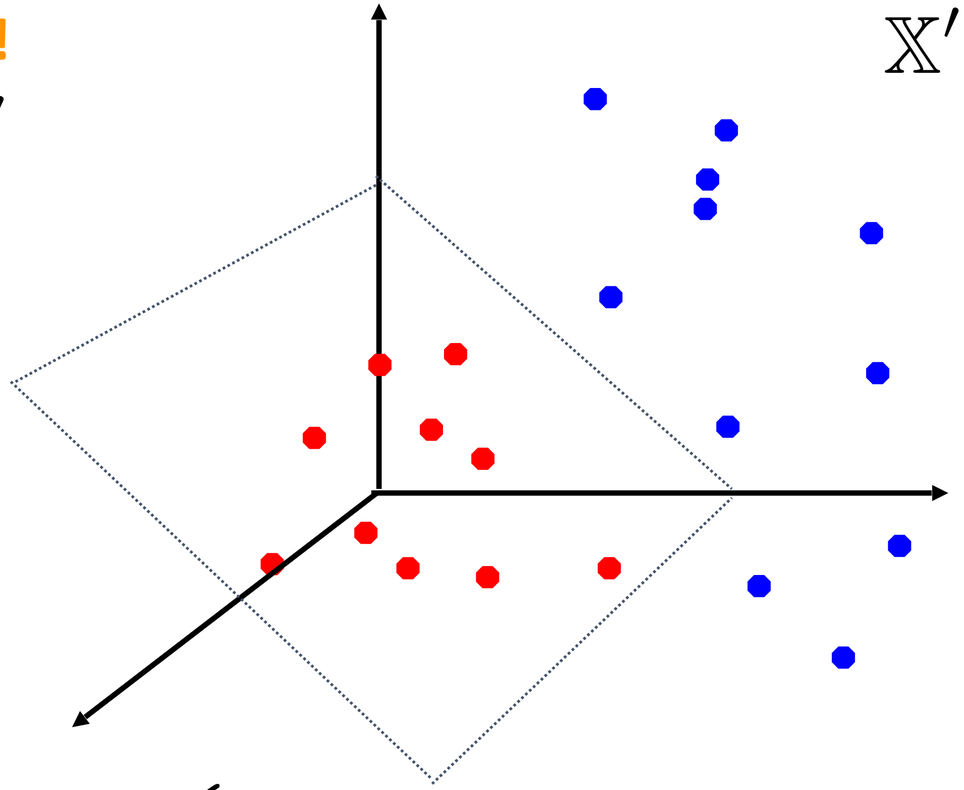
$$f_\theta(x) = \begin{cases} 1 & \text{if } w\ g(x) + b \geq 0 \\ 0 & \text{if } w\ g(x) + b < 0 \end{cases}$$

Reminder: Non-linear Decision Boundaries

This is what the hidden layers should be doing!

$$g : \mathbb{X} \longrightarrow \mathbb{X}'$$

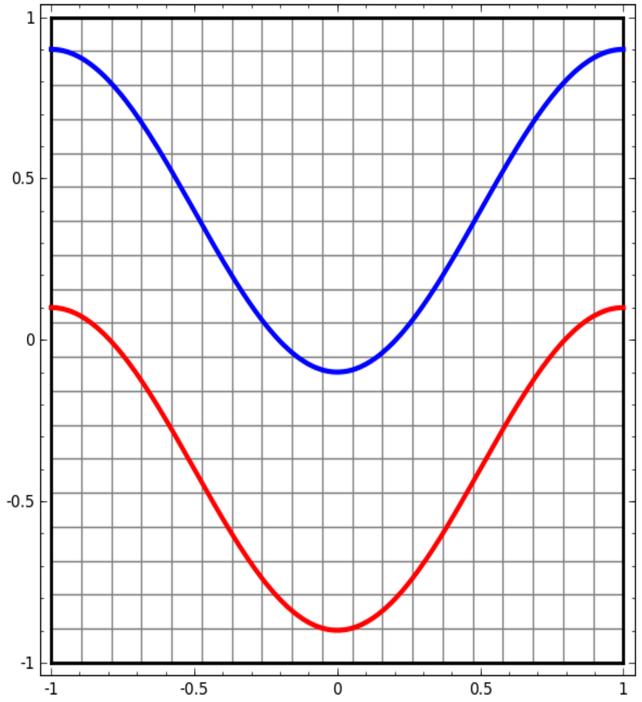




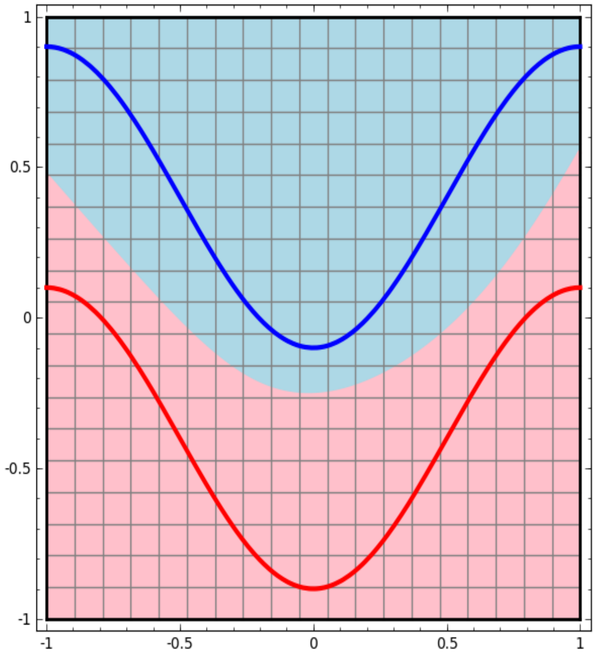
$$f_\theta(x) = \begin{cases} 1 & \text{if } w\ g(x) + b \geq 0 \\ 0 & \text{if } w\ g(x) + b < 0 \end{cases}$$

From Non-separable to Linearly Separable

Non-linearly separable data



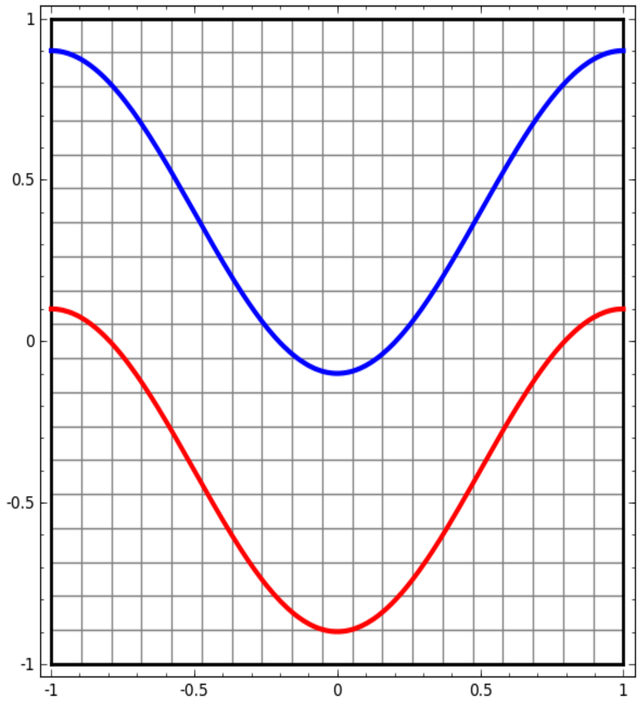
Decision function



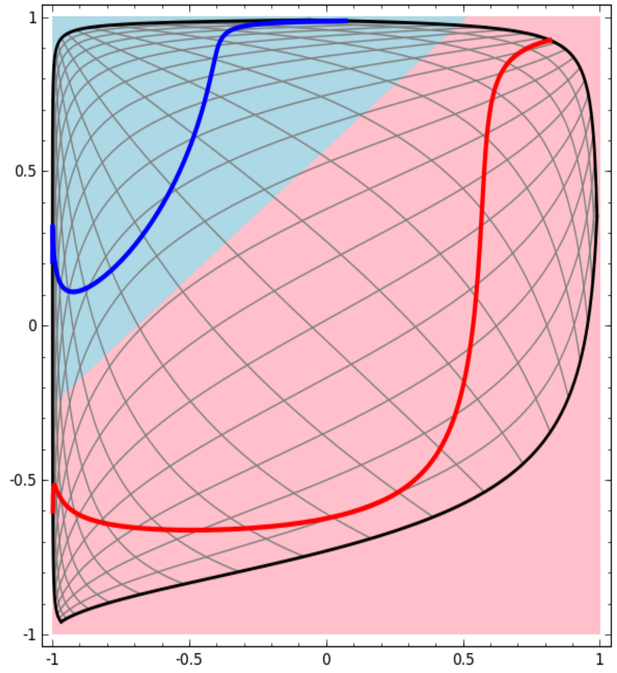
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

From Non-separable to Linearly Separable

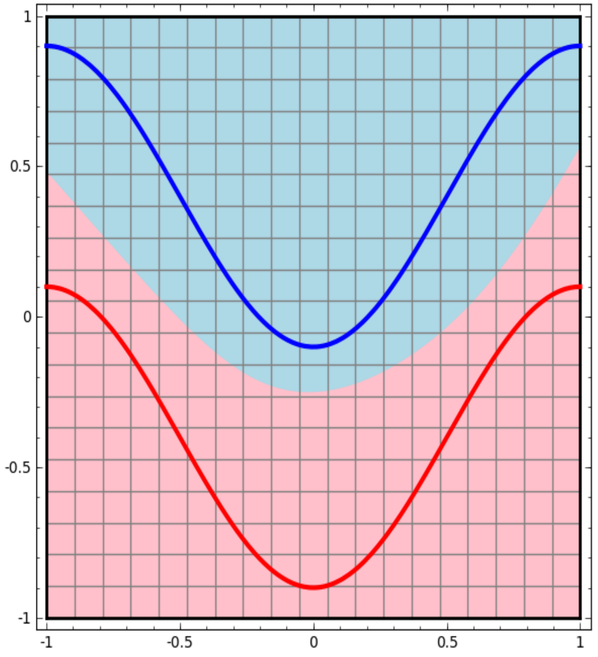
Non-linearly separable data



Data mapped to learned space



Decision function



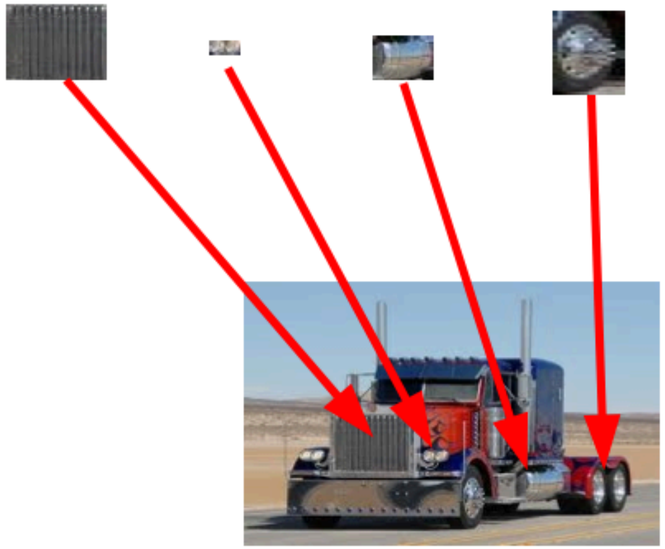
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Hidden Layers: What do They Do?

Intuition: learn “dictionary” for objects

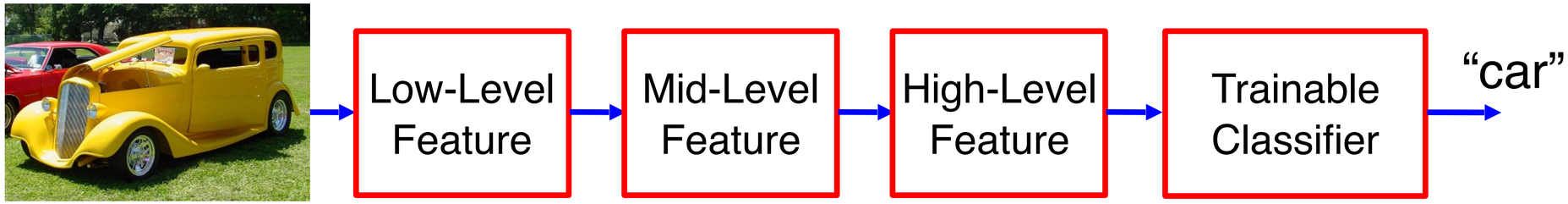
“Distributed representation”:
represent (and classify) objects by mixing & mashing reusable parts

[0 0 **1** 0 0 0 0 **1** 0 0 **1** **1** 0 0 **1** 0 ...] truck feature



Slide Credit: Marc'Aurelio Ranzato, Yann LeCun

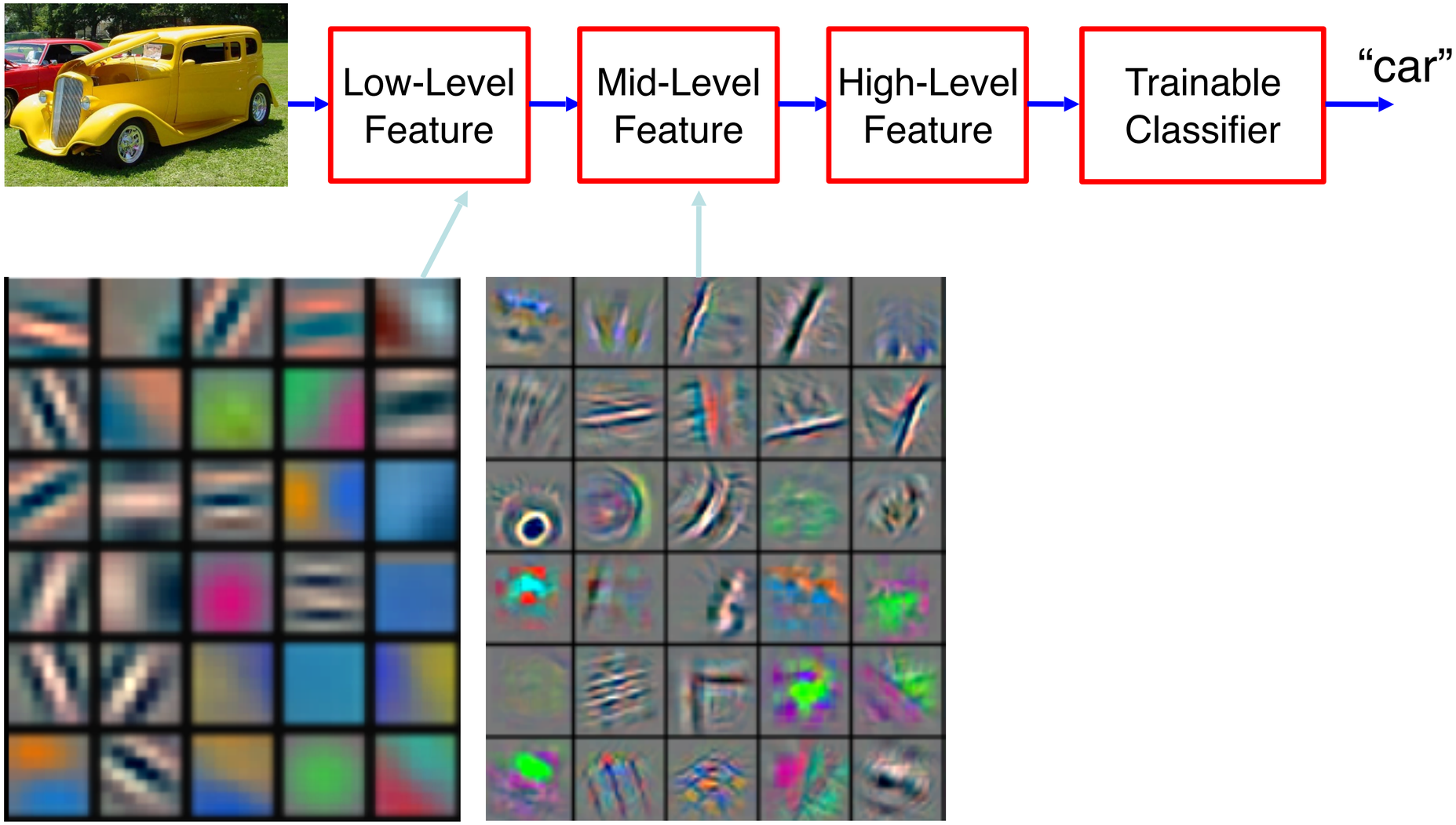
Deep Learning ~ Hierarchical Composition



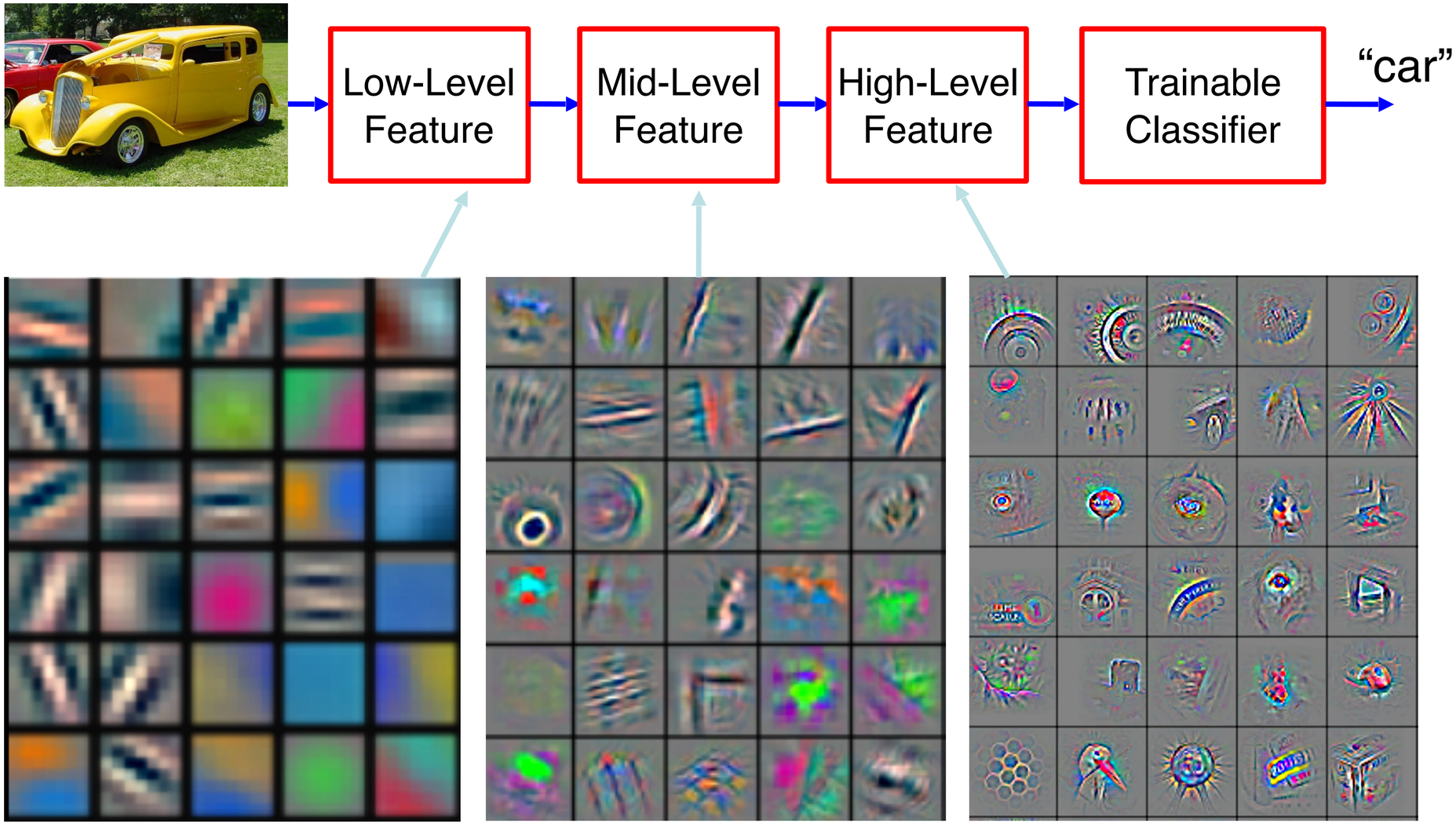
Deep Learning ~ Hierarchical Composition

Low-Level Feature → Mid-Level Feature → High-Level Feature → Trainable Classifier → “car”

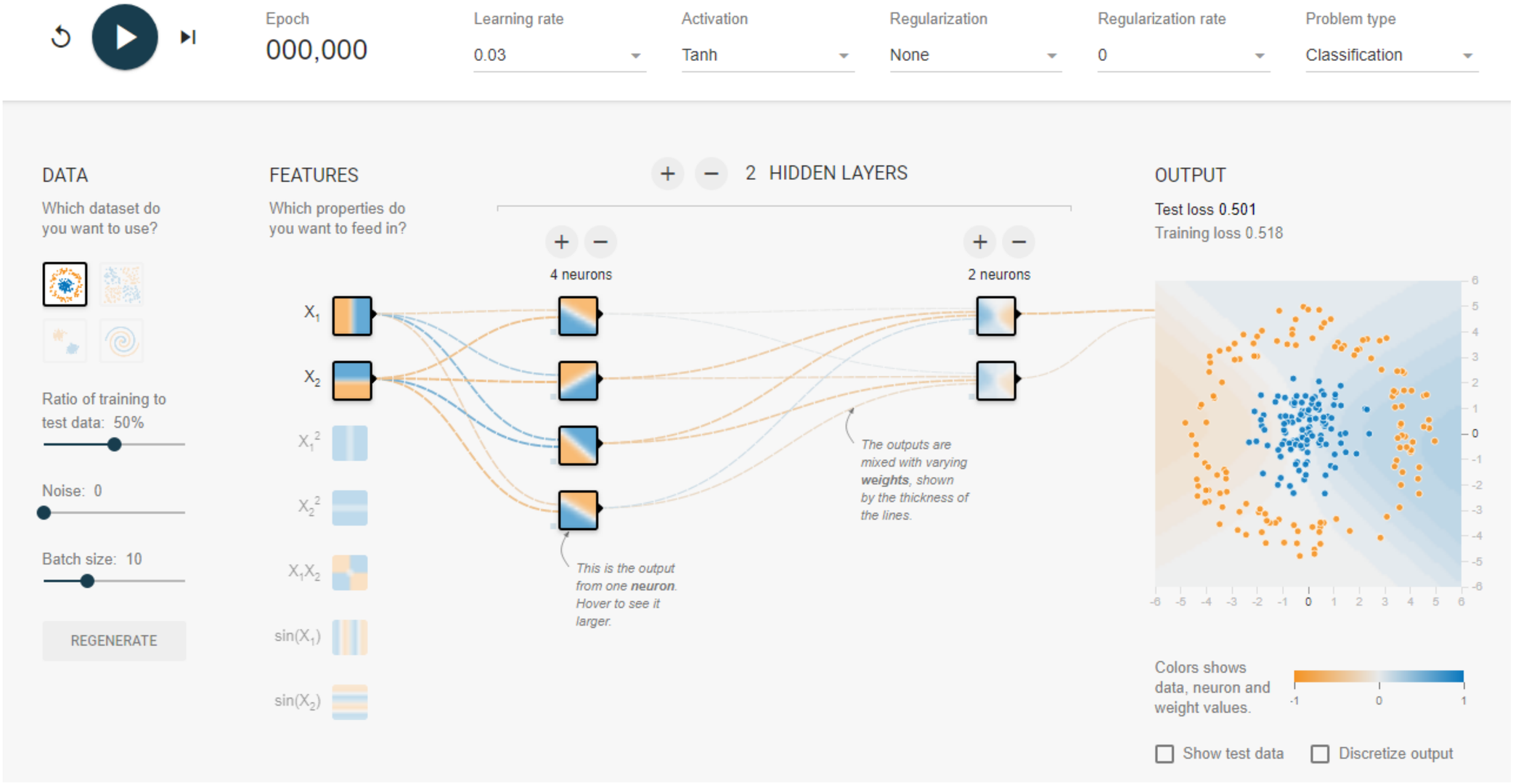
Deep Learning ~ Hierarchical Composition



Deep Learning ~ Hierarchical Composition



MLP Demo: playground.tensorflow.org



Neural Network Training: Old and New Tricks

- Old
  - **Backpropagation algorithm**
  - Stochastic gradient, momentum, weight decay

- New
  - Dropout
  - Relu
  - Batch Norm(alization), GroupNorm, Spectral Normalization
  - Res(idual) Net(work)

Training Goal

Our network implements a parametric function:

$$f_\theta : \mathbb{X} \longrightarrow \mathbb{Y} \qquad \hat{y} = f(x;\theta)$$

Training Goal

Our network implements a parametric function:

$$f_\theta : \mathbb{X} \longrightarrow \mathbb{Y} \qquad \hat{y} = f(x;\theta)$$

During training, we search for parameters that minimize a loss:

$$\min_\theta L_f(\theta)$$

Training Goal

Our network implements a parametric function:

$$f_\theta : \mathbb{X} \longrightarrow \mathbb{Y} \qquad \hat{y} = f(x;\theta)$$

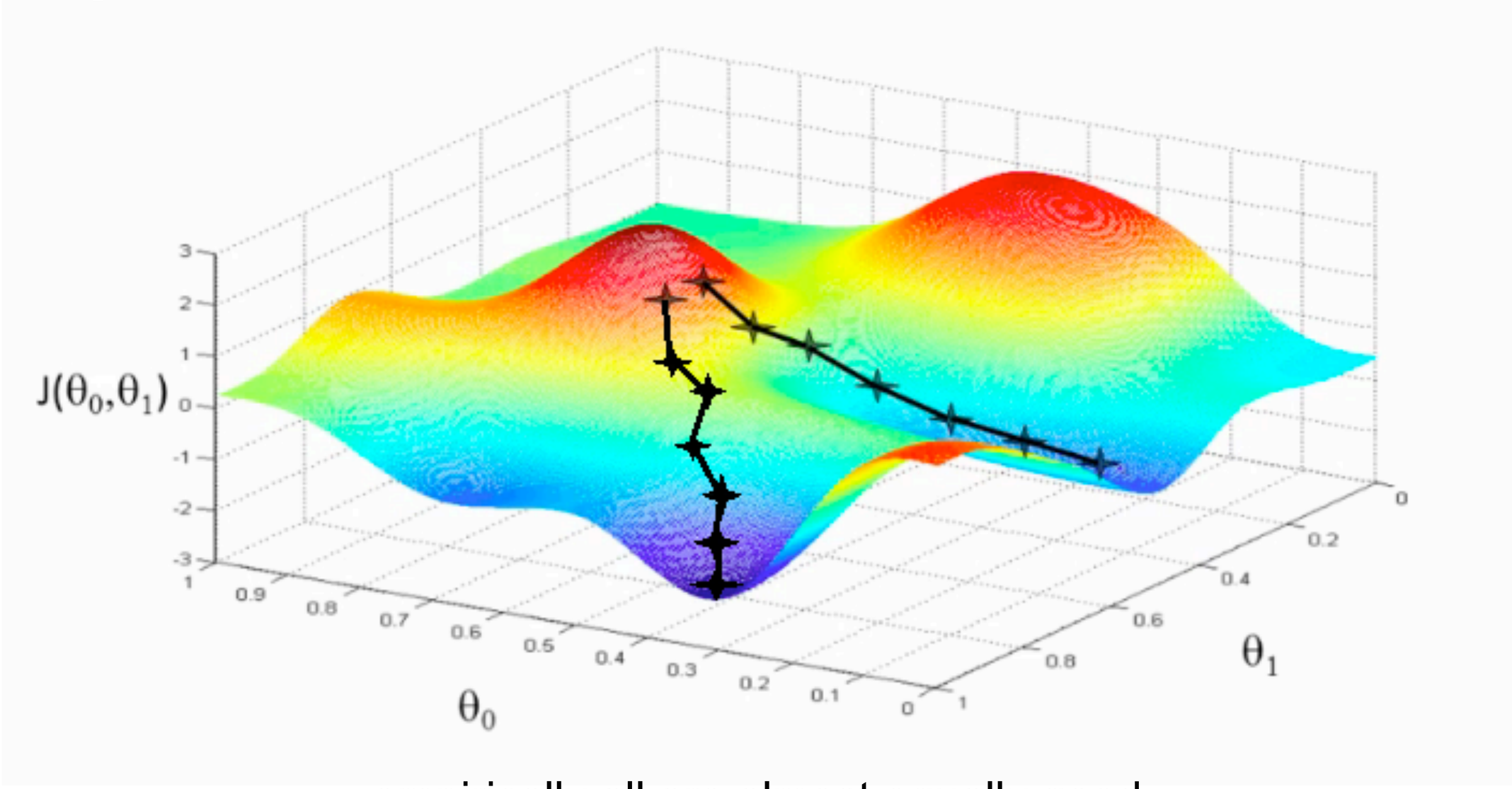
During training, we search for parameters that minimize a loss:

$$\min_\theta L_f(\theta)$$

Example: L2 regression loss given target (x^i, y^i) pairs :

$$L_f(\theta) = \sum_i \| f(x^i;\theta) - y^i \|_2^2$$

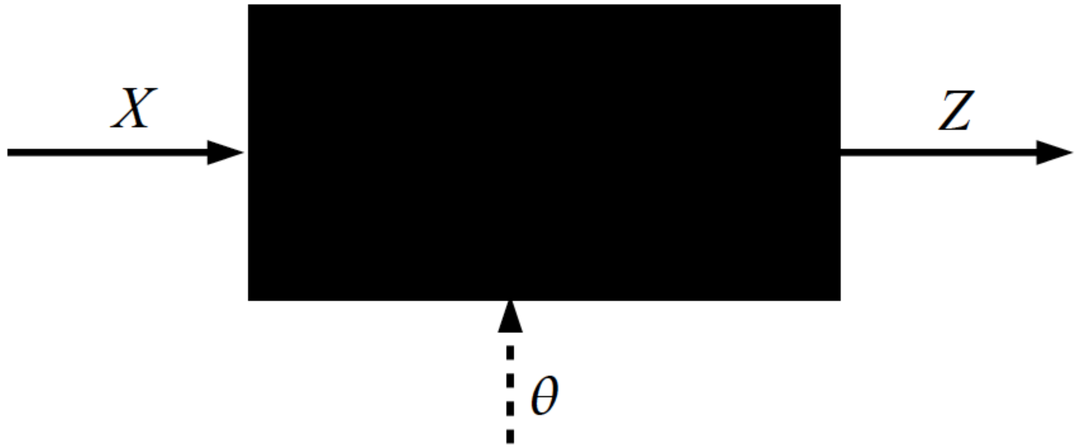
Multiple Local Minima: Based on Initialization



empirically all are almost equally good

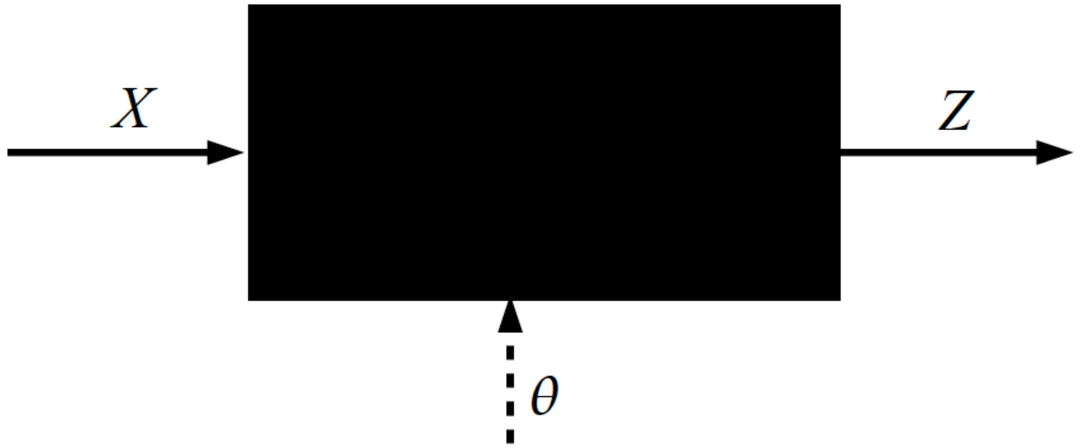
All You Need is Gradients

Forward

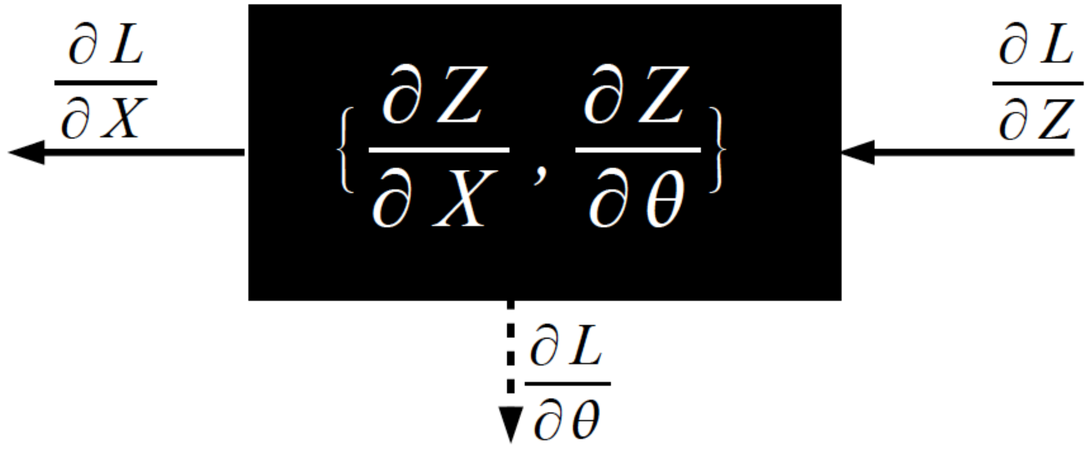


All You Need is Gradients

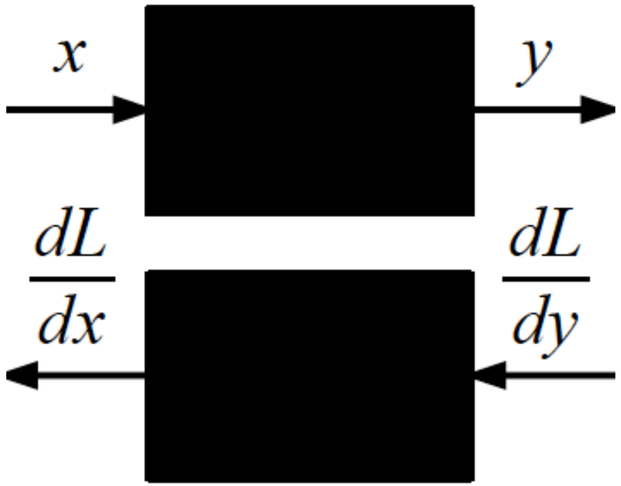
Forward



Backward



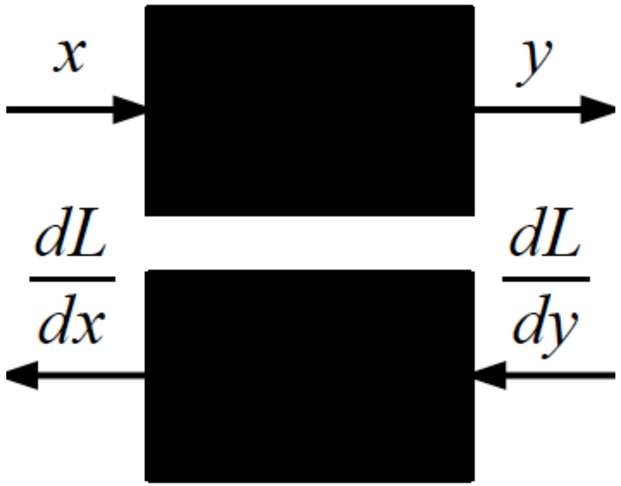
Chain Rule



Given $y(x)$ and dL/dy,

What is dL/dx ?

Chain Rule

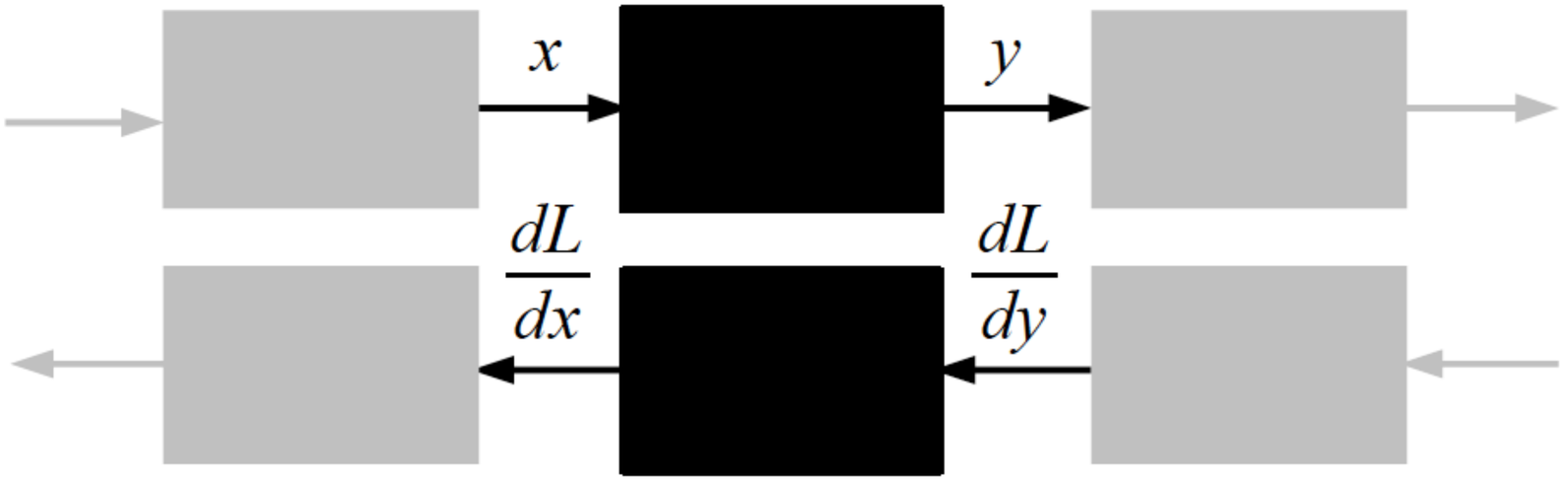


Given $y(x)$ and dL/dy,

What is dL/dx ?

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$$

‘Another Brick in the Wall’

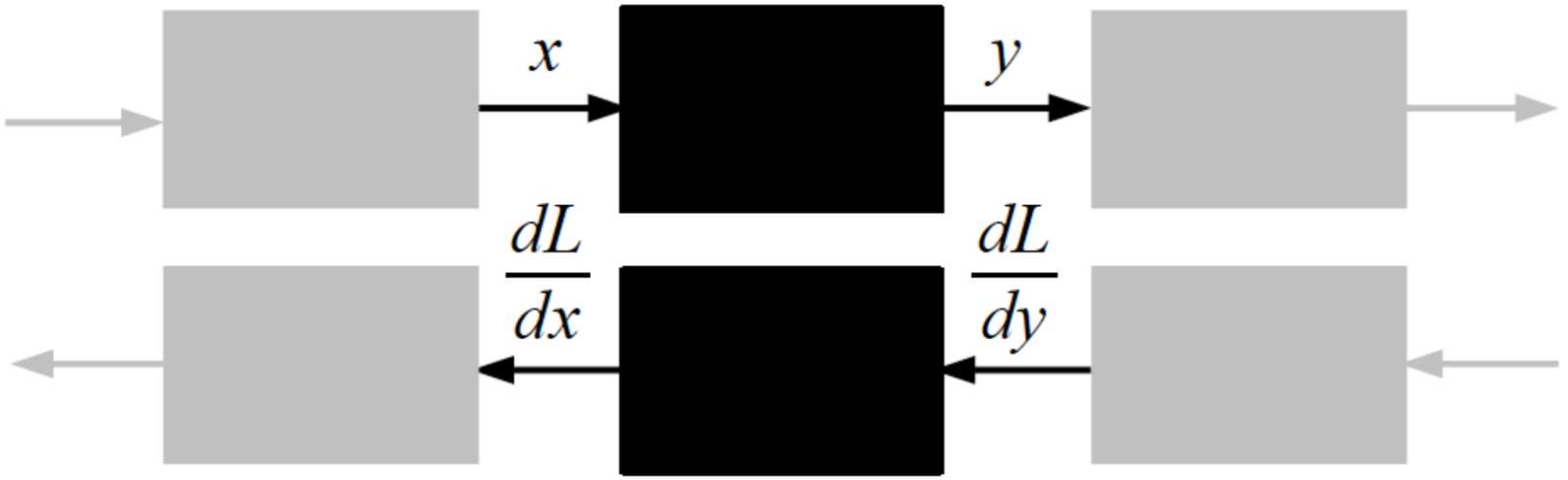


Given $y(x)$ and dL/dy,

What is dL/dx ?

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$$

‘Another Brick in the Wall’



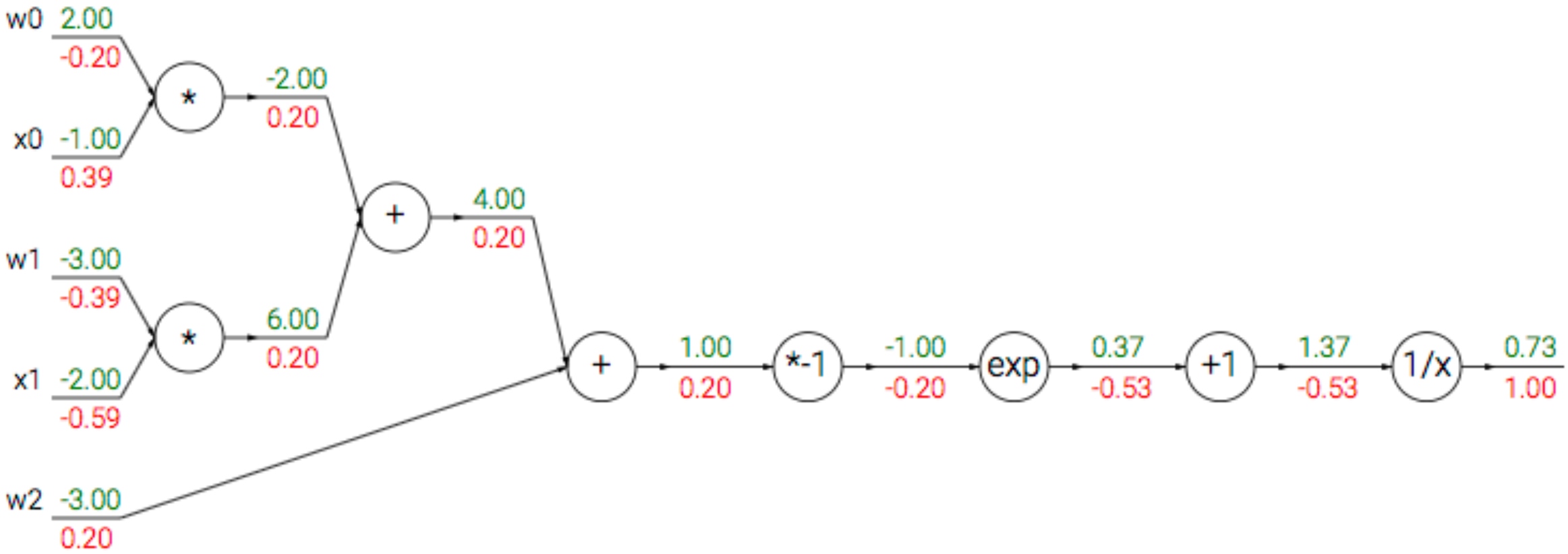
Given $y(x)$ and dL/dy,

What is dL/dx ?

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$$

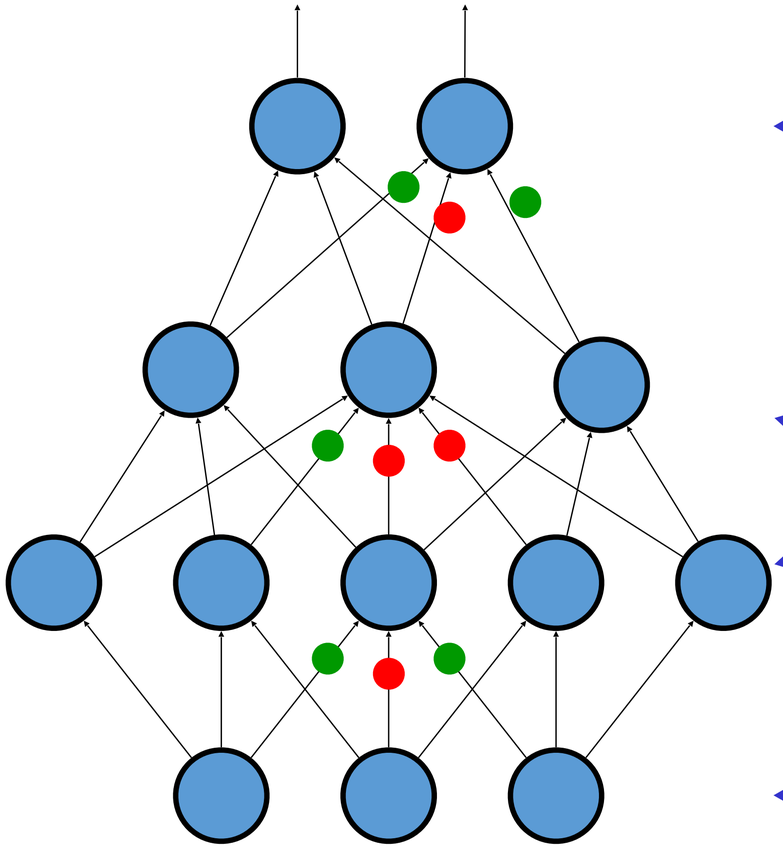
Deep Lea

Computation Graph and Automatic Differentiation



Multi-Layer Perceptrons (~1985)

$$u_i = g\left(\sum_{k\in\mathcal{N}(i)} w_{k,i}\, g\left(\sum_{m\in\mathcal{N}(k)} w_{m,k}u_m + b_k\right) + b_i\right)$$



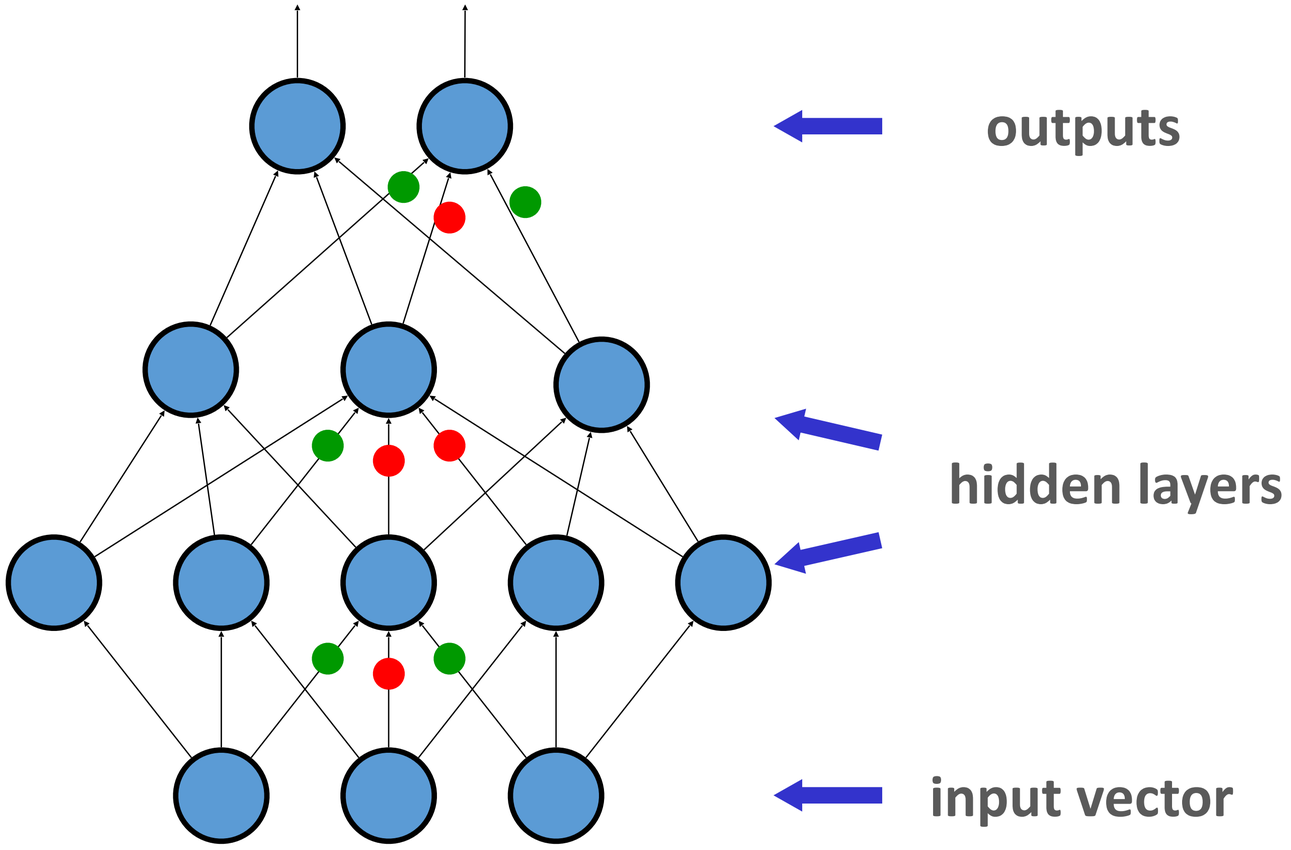
outputs

hidden layers

input vector

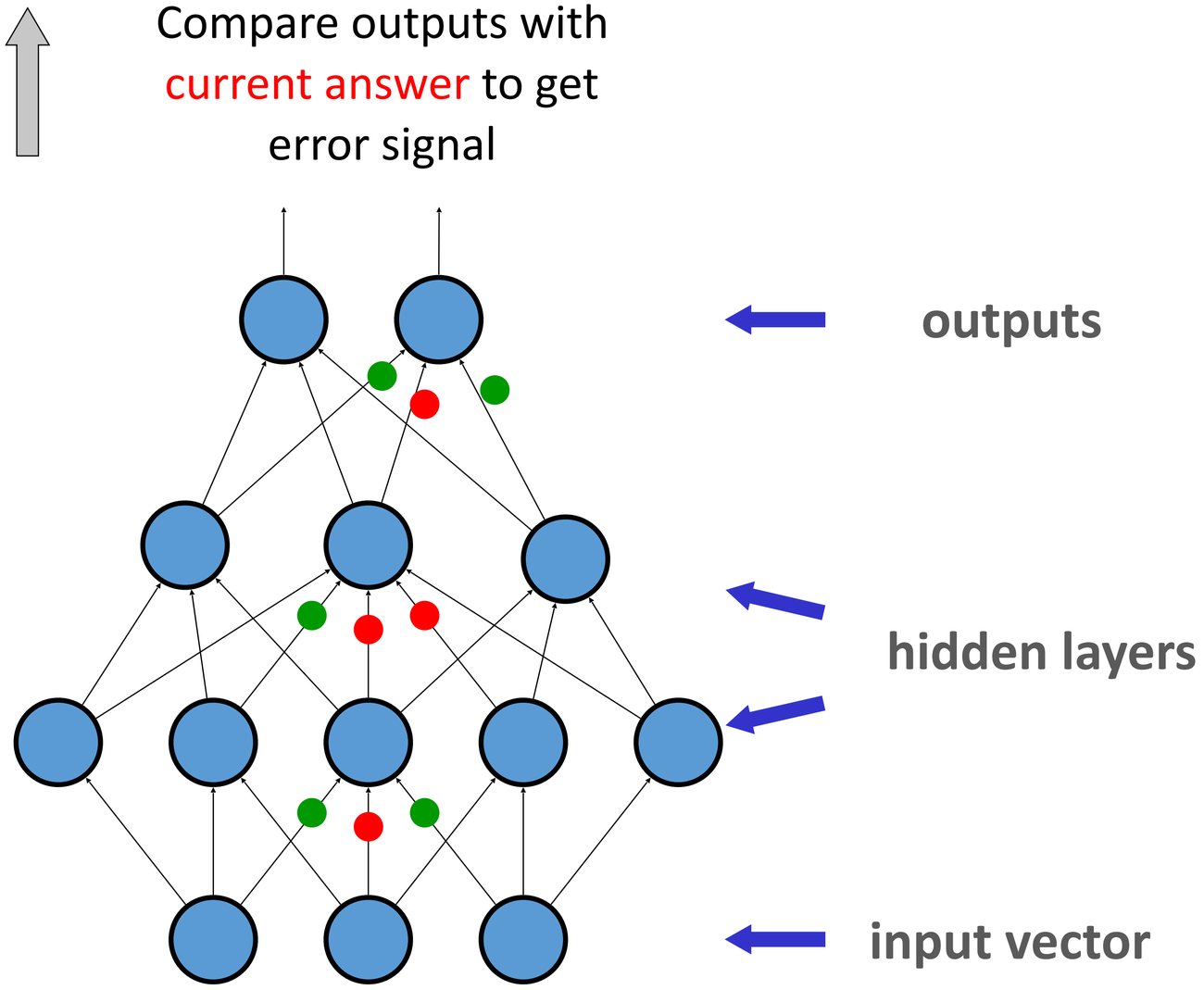
Slide credit: G. Hinton

Multi-Layer Perceptrons (~1985)



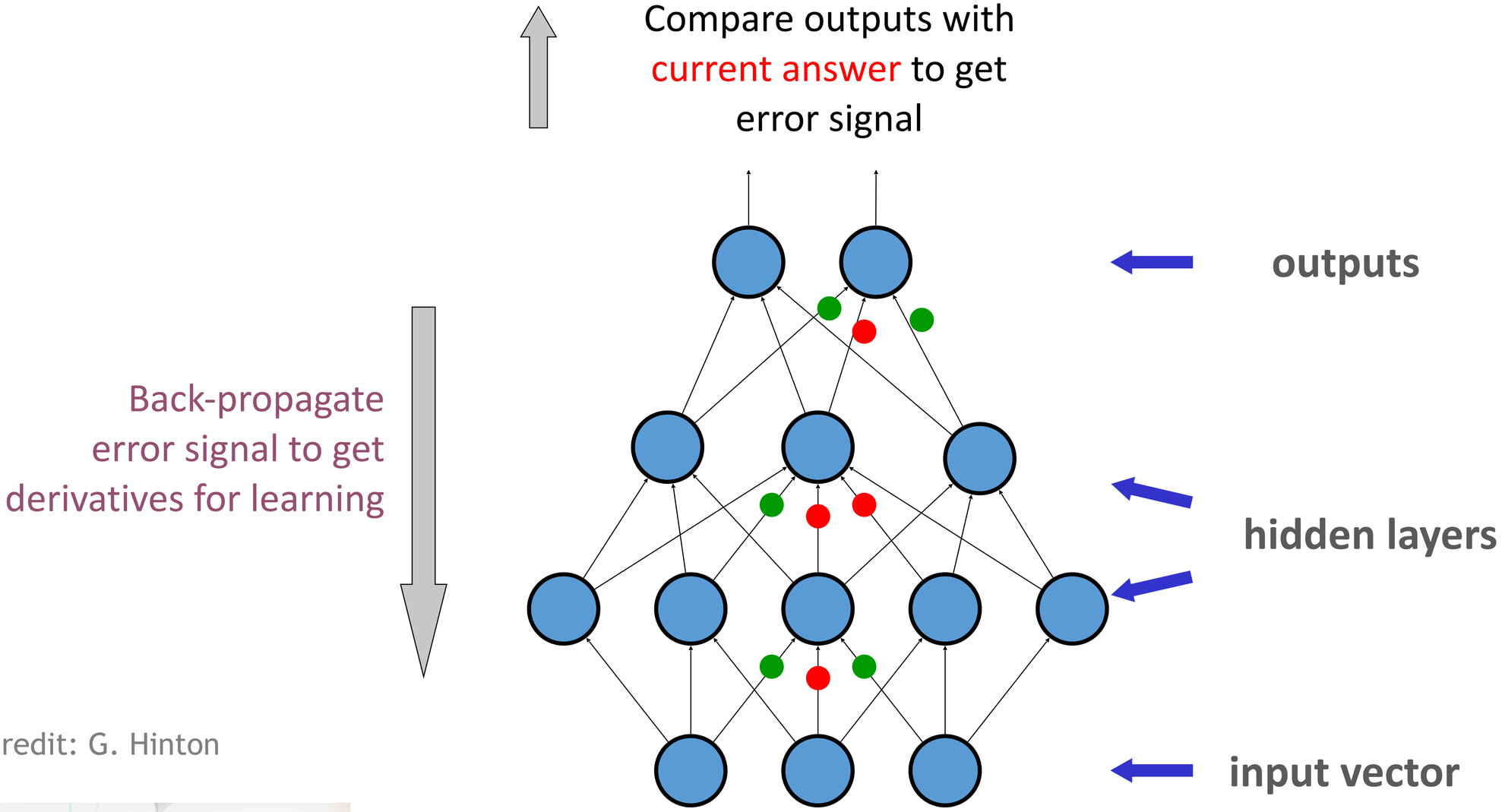
Slide credit: G. Hinton

Multi-Layer Perceptrons (~1985)



Slide credit: G. Hinton

Multi-Layer Perceptrons (~1985)



Slide credit: G. Hinton

Training Goal

Our network implements a parametric function:

$$f_\theta : \mathbb{X} \longrightarrow \mathbb{Y} \qquad \hat{y} = f(x;\theta)$$

During training, we search for parameters that minimize a loss:

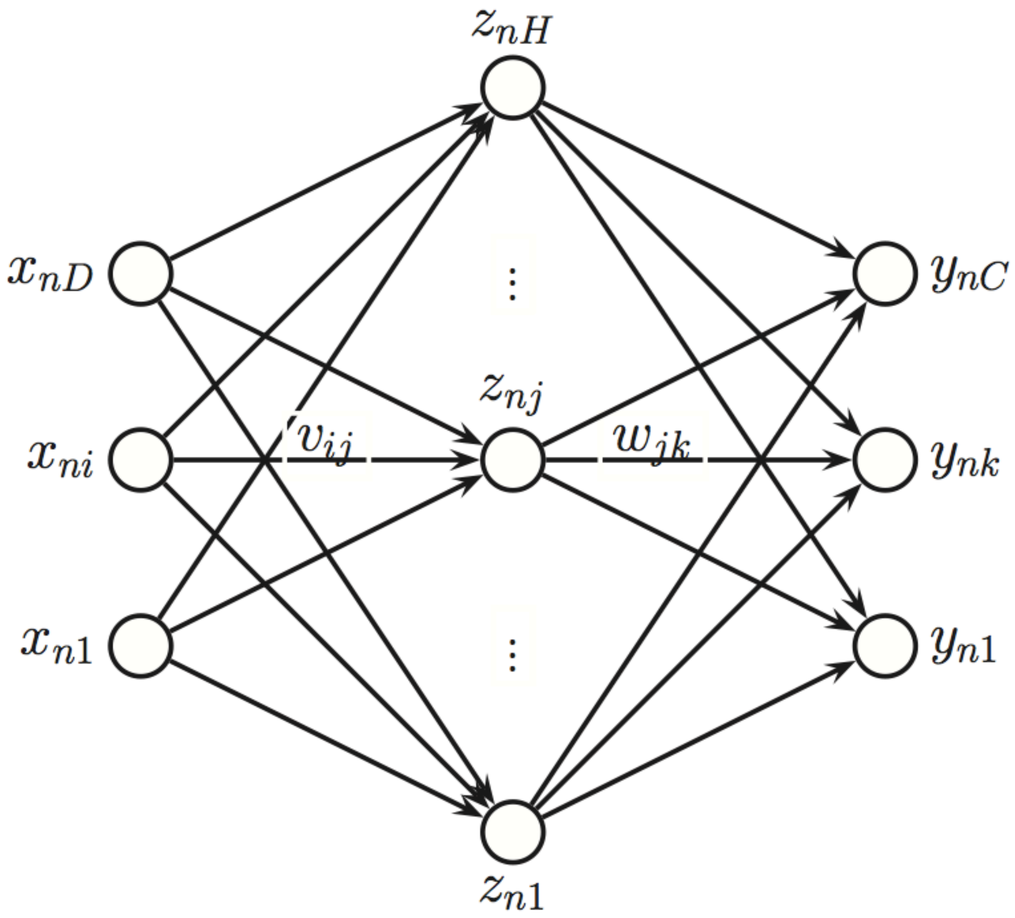
$$\min_\theta L_f(\theta)$$

Example: L2 regression loss given target (x^i, y^i) pairs :

$$L_f(\theta) = \sum_i \| f(x^i;\theta) - y^i \|_2^2$$

A Neural Network for Multi-way Classification

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n$$



Inputs

Outputs

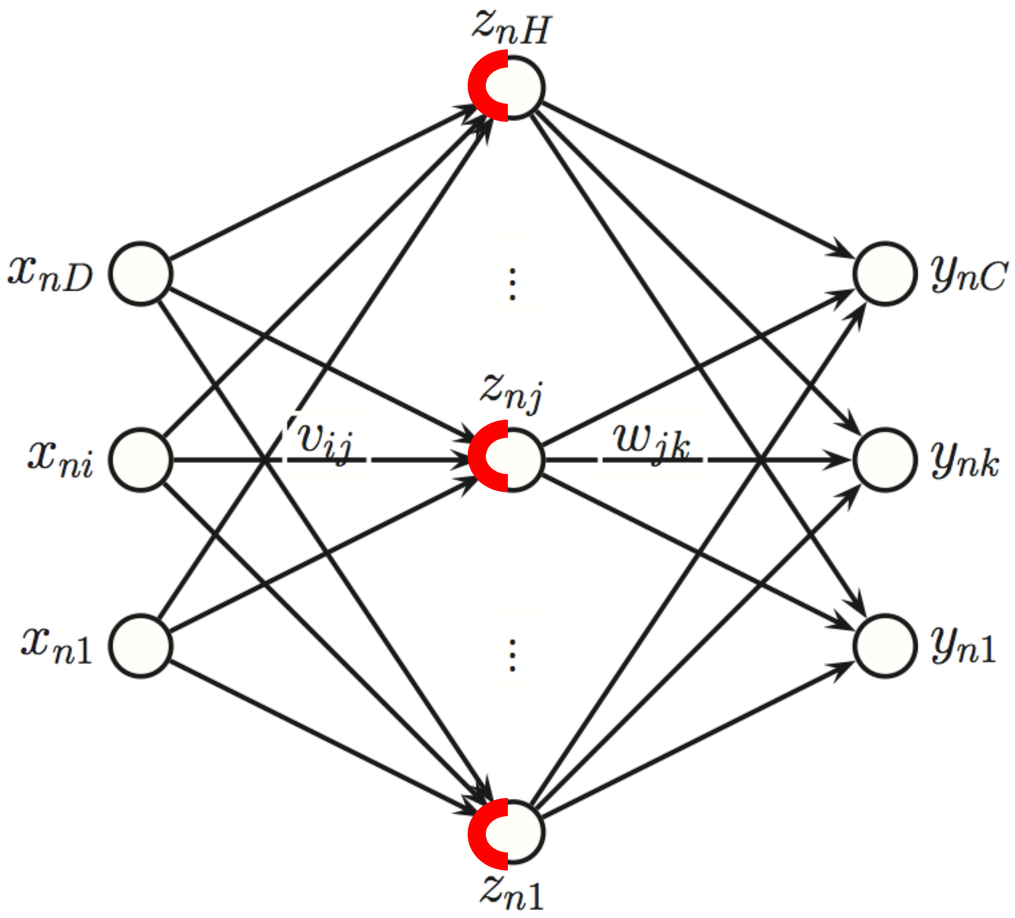
Hidden layer

Parameters:

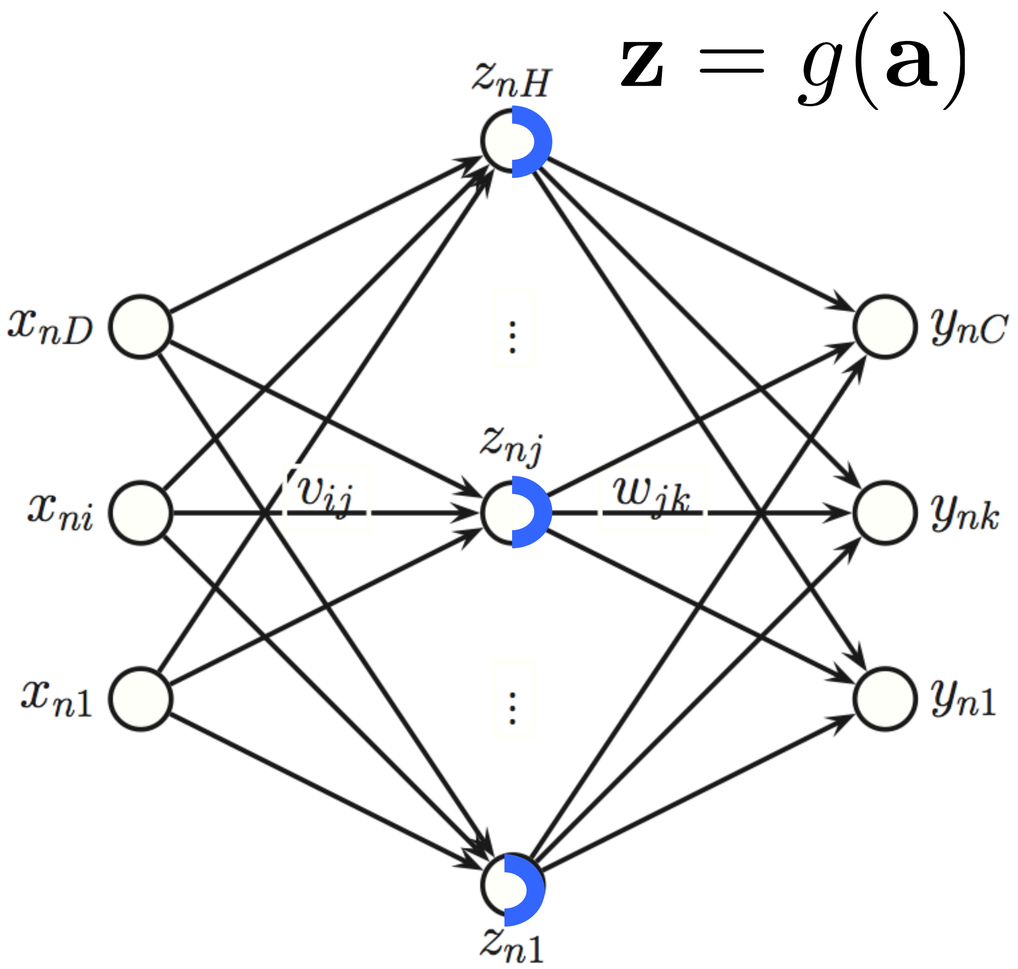
$$\theta = \{\mathbf{v}, \mathbf{w}\}$$

A Neural Network in Forward Mode

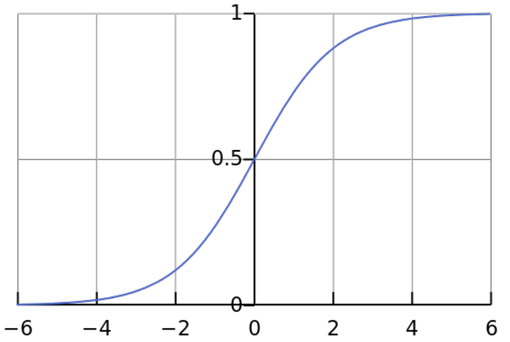
$$\mathbf{a} = \mathbf{V}\mathbf{x}$$



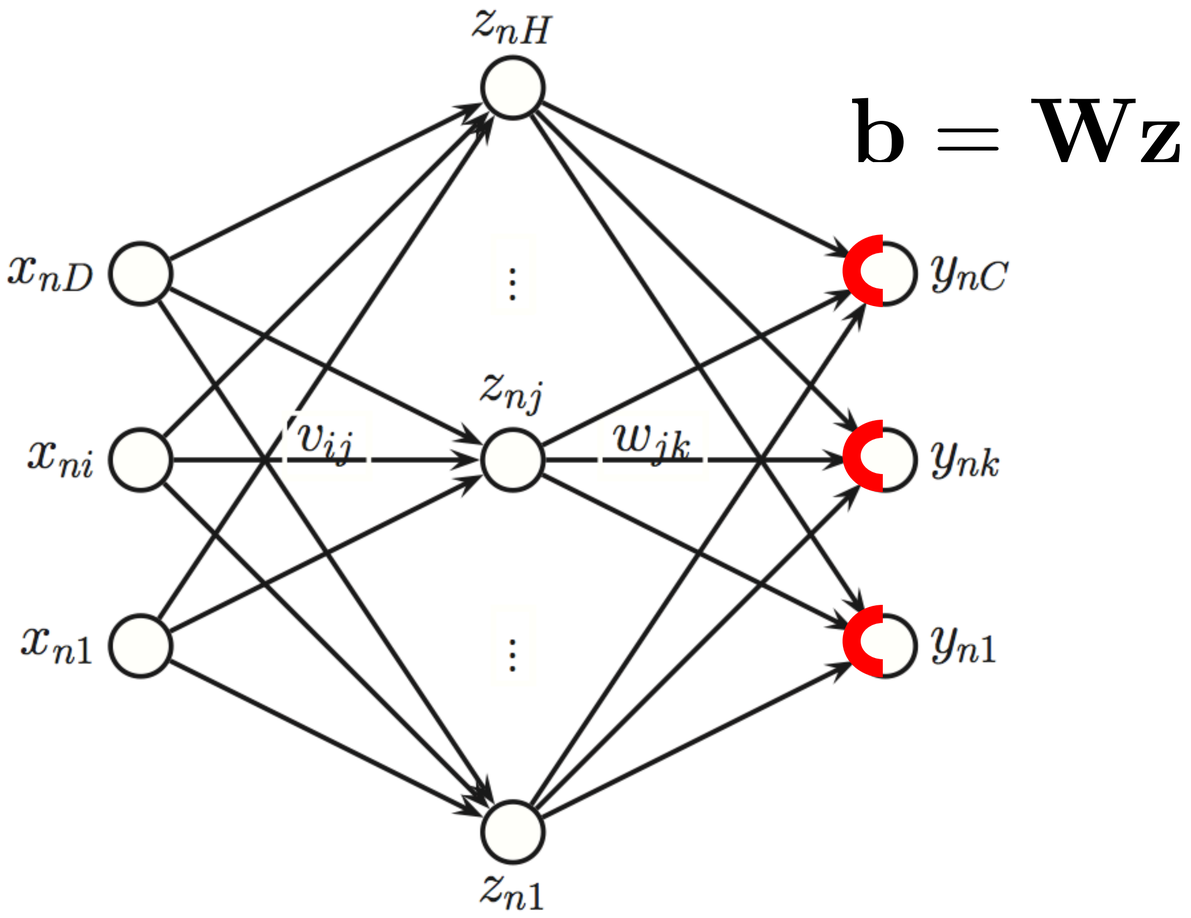
A Neural Network in Forward Mode



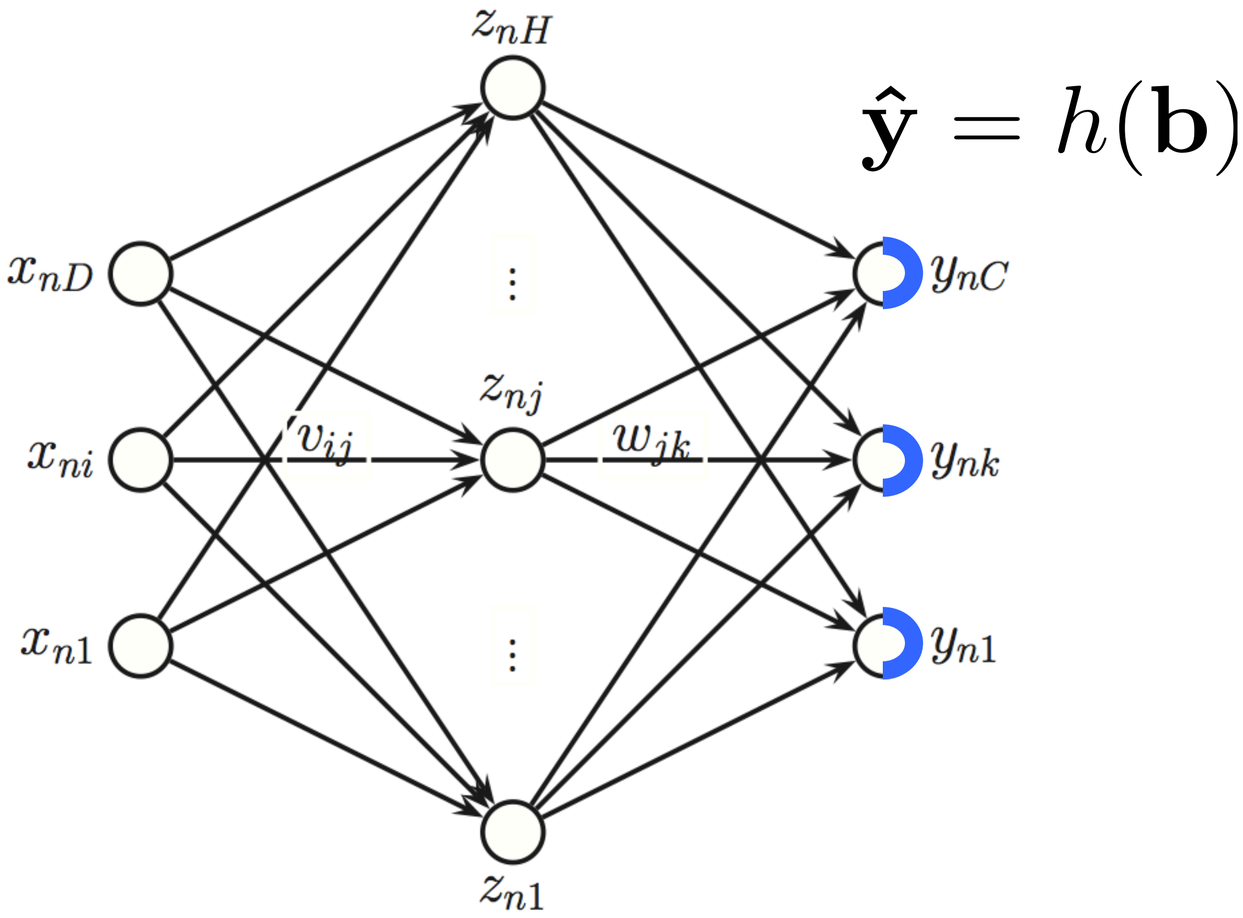
$$z_k = \frac{1}{1 + \exp(-a_k)}$$



A Neural Network in Forward Mode

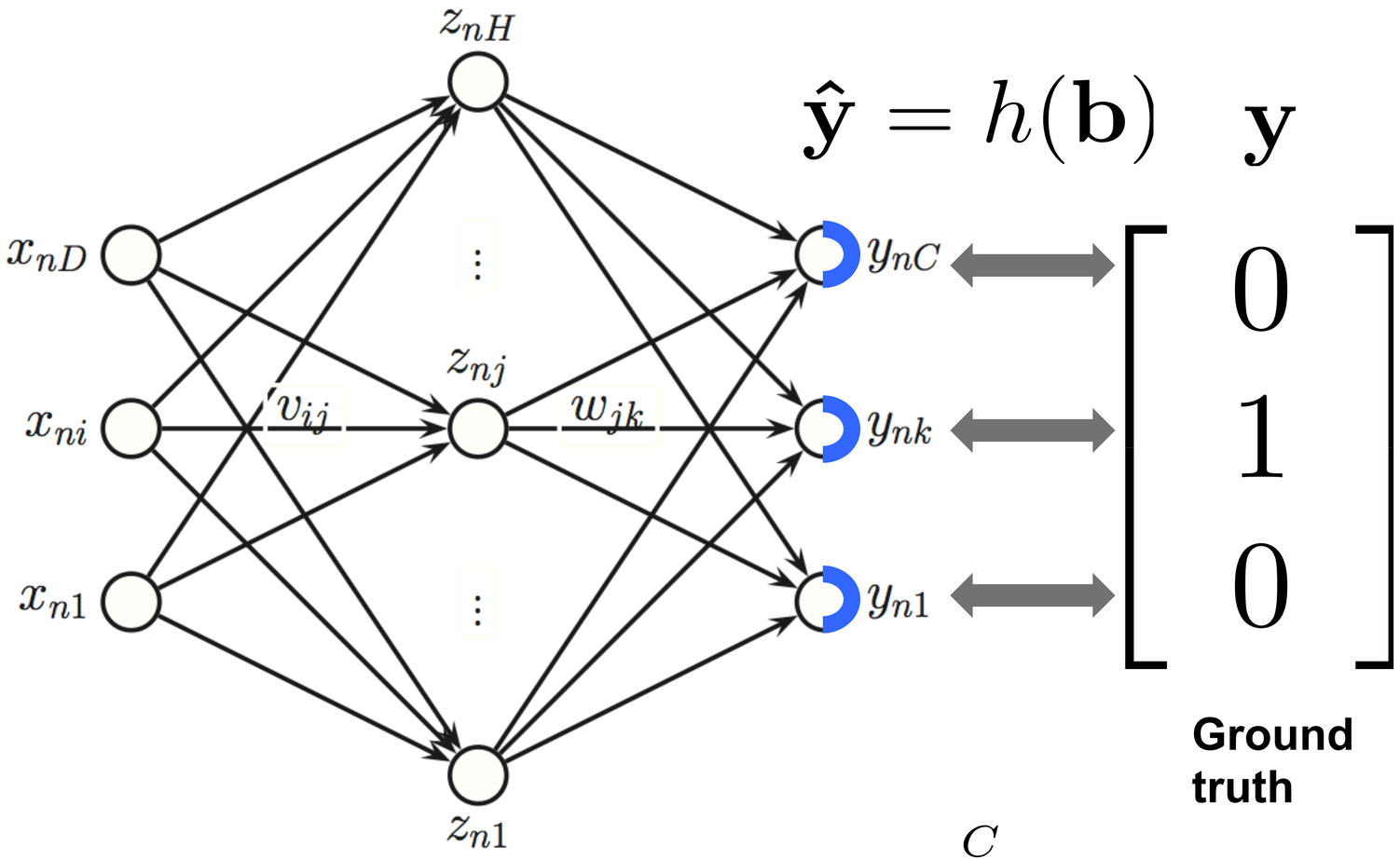


A Neural Network in Forward Mode



Objective for Linear Regression

$$h(\mathbf{b}) = \mathbf{b}$$



$$\hat{\mathbf{y}} = h(\mathbf{b}) \quad \mathbf{y}$$

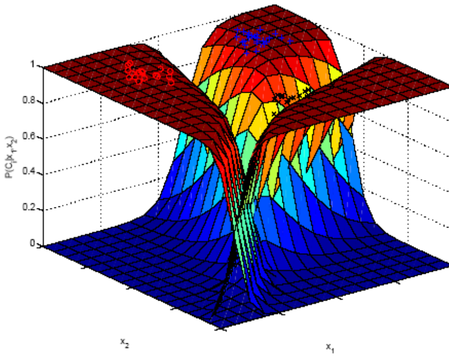
$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Ground truth

L2 loss

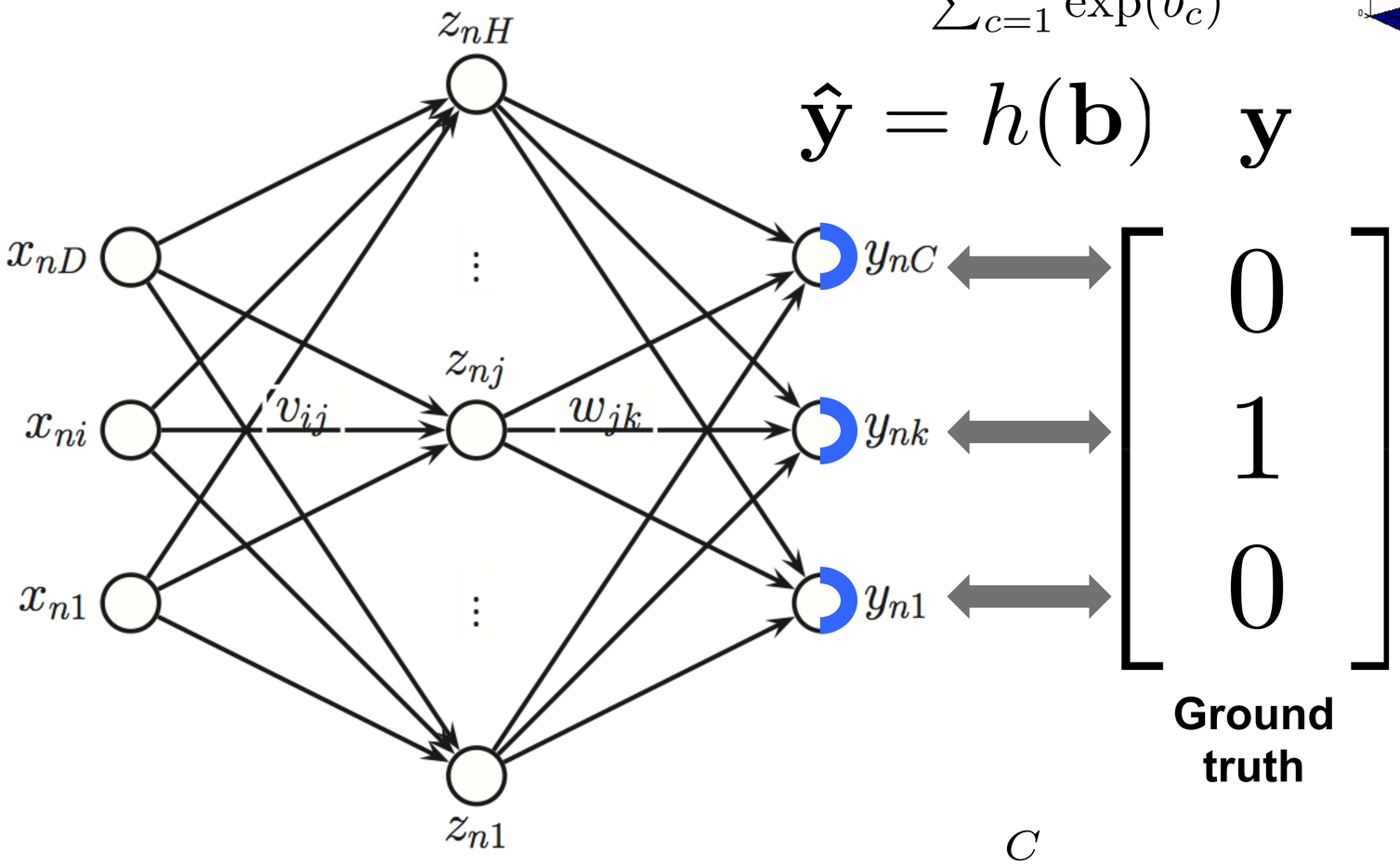
$$l(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{c=1}^{C} (\mathbf{y}_c - \hat{\mathbf{y}}_c)^2$$

Objective for Multi-class Classification



Softmax unit $\hat{y}_k = \frac{\exp(b_k)}{\sum_{c=1}^{C} \exp(b_c)}$

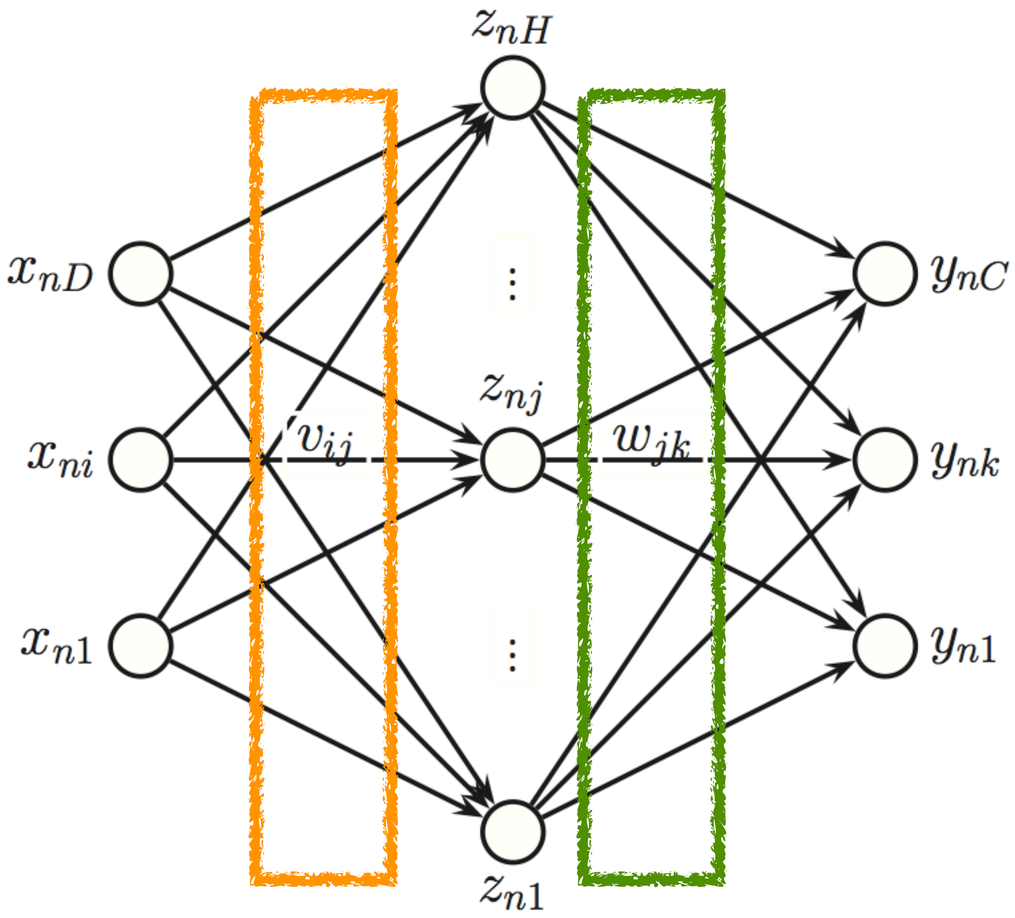
$\hat{\mathbf{y}} = h(\mathbf{b})$ $\mathbf{y}$



'Cross-entropy' loss $l(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{c=1}^{C} \mathbf{y}_c \log(\hat{\mathbf{y}}_c)$

A Neural Network for Multi-way Classification

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n$$



Inputs

Outputs

Hidden layer

Parameters:

$$\theta = \{\mathbf{v}, \mathbf{w}\}$$

Neural Network in Forward Mode: Recap

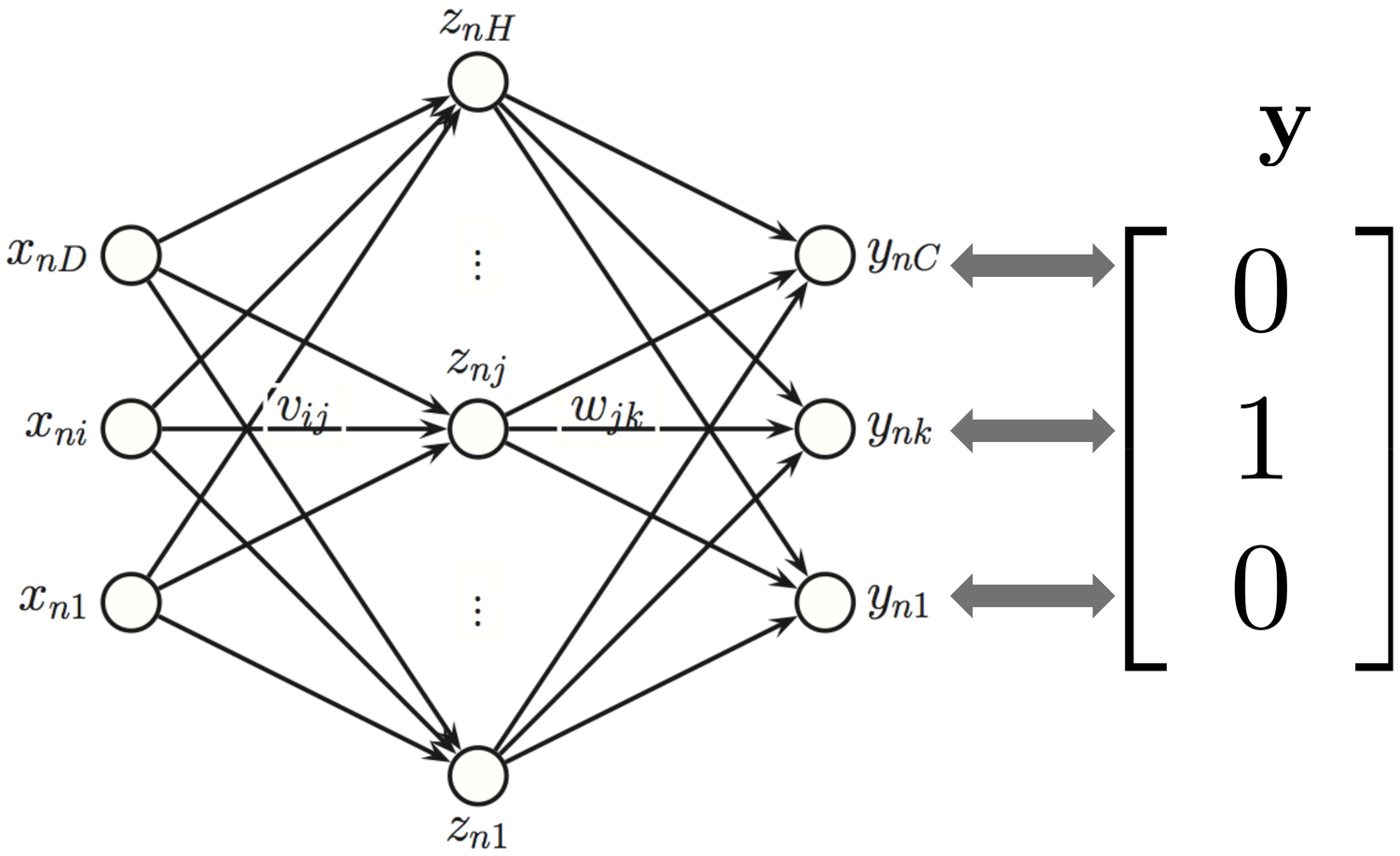
Network output: $\hat{\mathbf{y}} = f(\mathbf{x}; \mathbf{v}, \mathbf{w})$

Loss (prediction error): $l(\hat{\mathbf{y}}, \mathbf{y})$

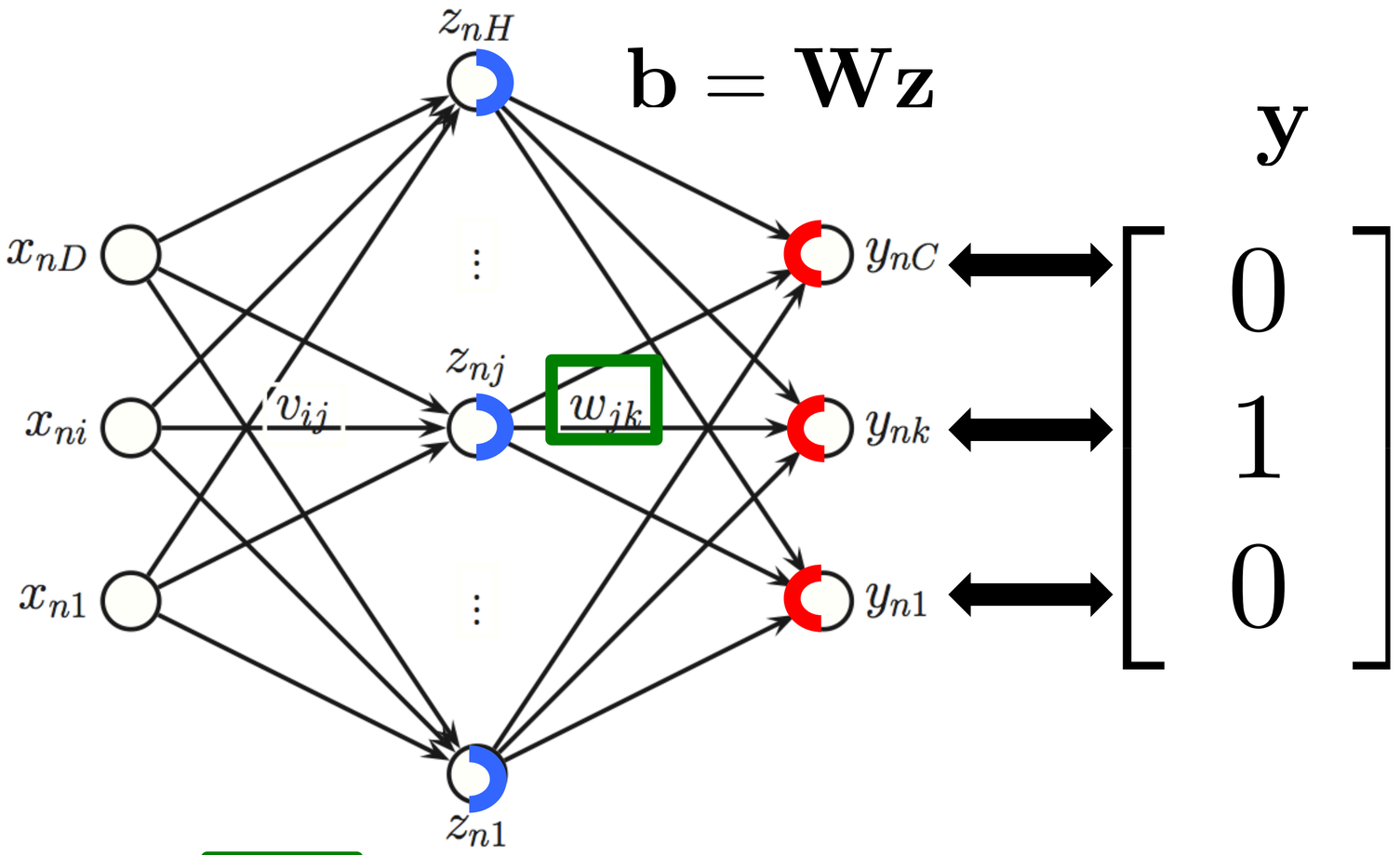
What we need to compute for gradient descent: $\frac{\partial l(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{v}_i}$

$\frac{\partial l(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{w}_j}$

A Neural Network in Backward Mode

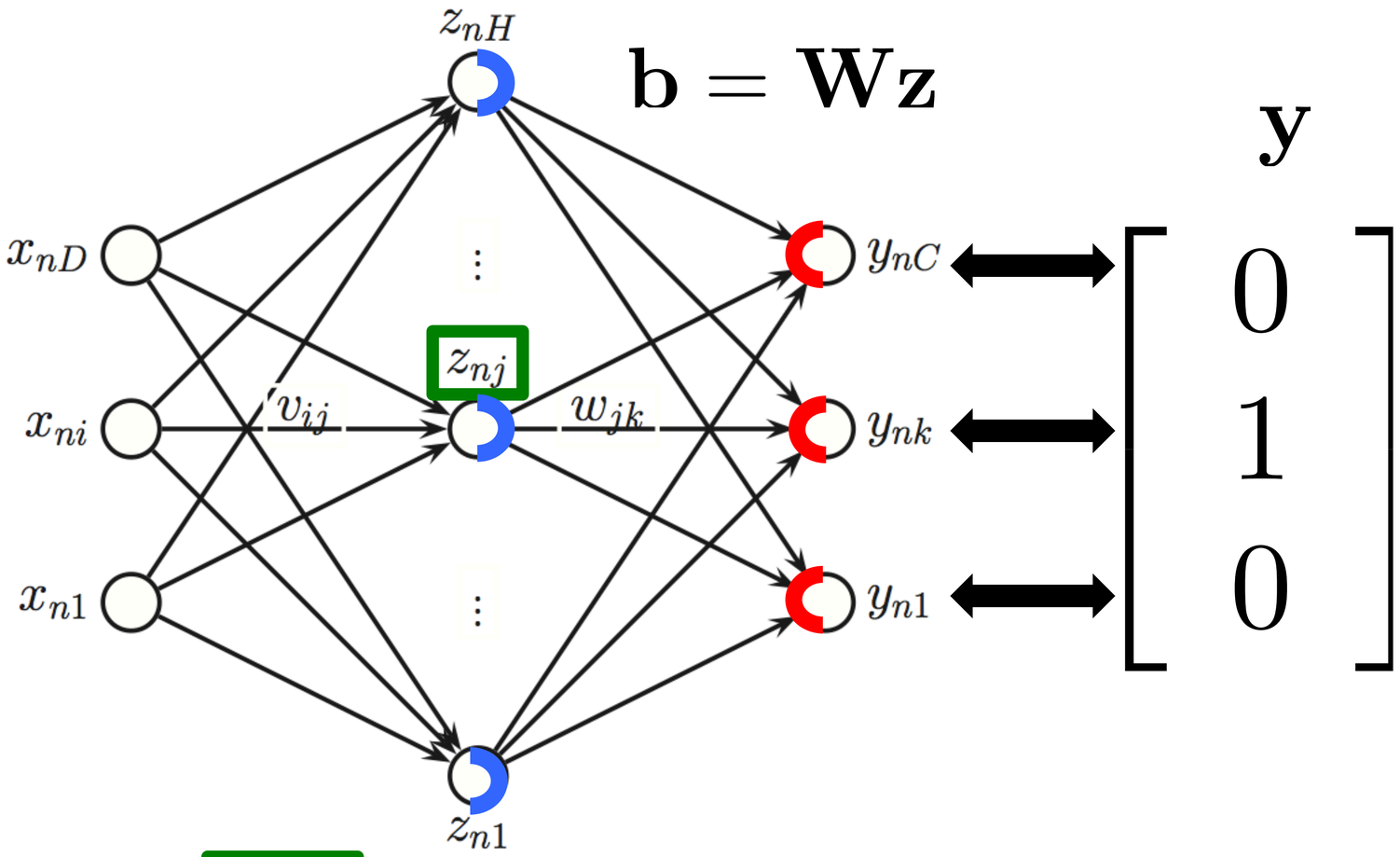


A Neural Network in Backward Mode



This we want $\frac{\partial l}{\partial w_{jk}} = ?$

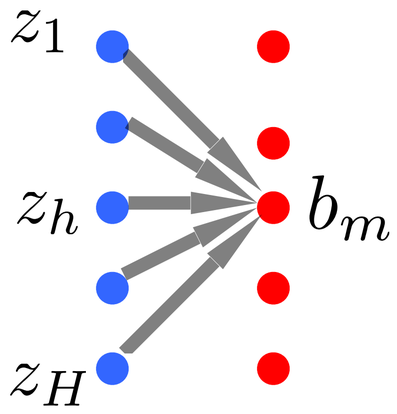
A Neural Network in Backward Mode



This we want $\boxed{\frac{\partial l}{\partial z_j}} = ?$

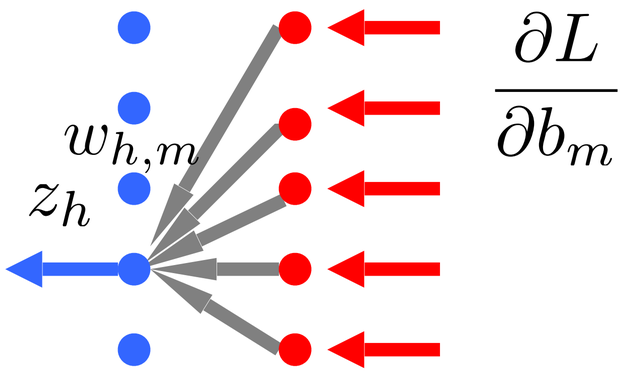
Linear Layer in Forward Mode: All For One

$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



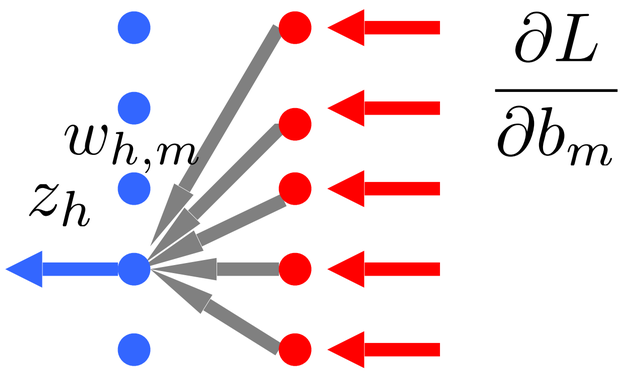
Linear Layer in Backward Mode: All For One

$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



Linear Layer in Backward Mode: All For One

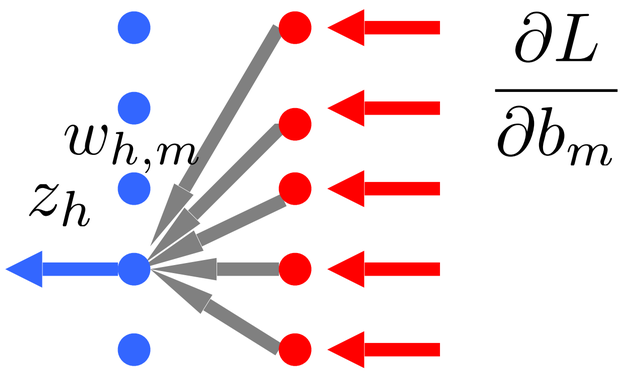
$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



$$\frac{\partial L}{\partial z_h} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial z_h}$$

Linear Layer in Backward Mode: All For One

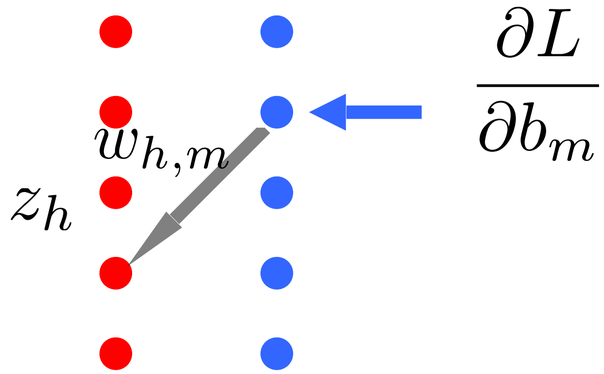
$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



$$\frac{\partial L}{\partial z_h} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial z_h} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} w_{h,c}$$

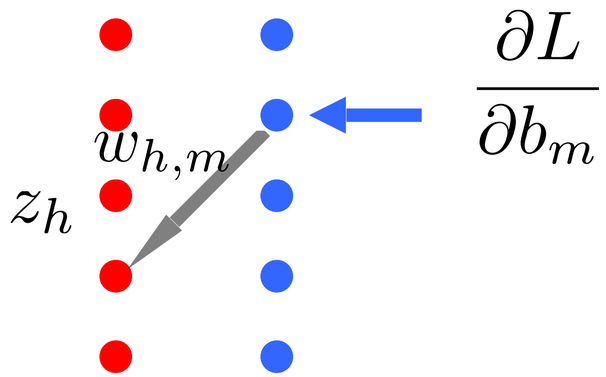
Linear Layer Parameters in Backward: 1-to-1

$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



Linear Layer Parameters in Backward: 1-to-1

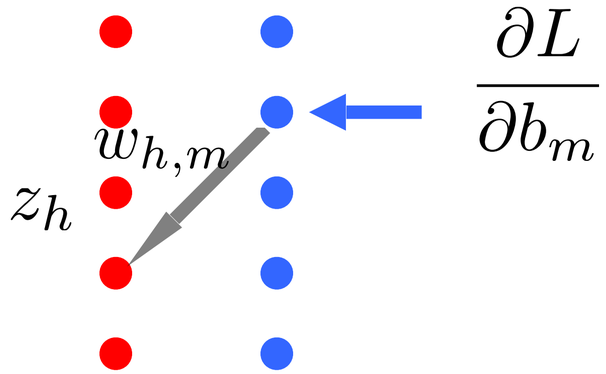
$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



$$\frac{\partial L}{\partial w_{h,m}} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial w_{h,m}}$$

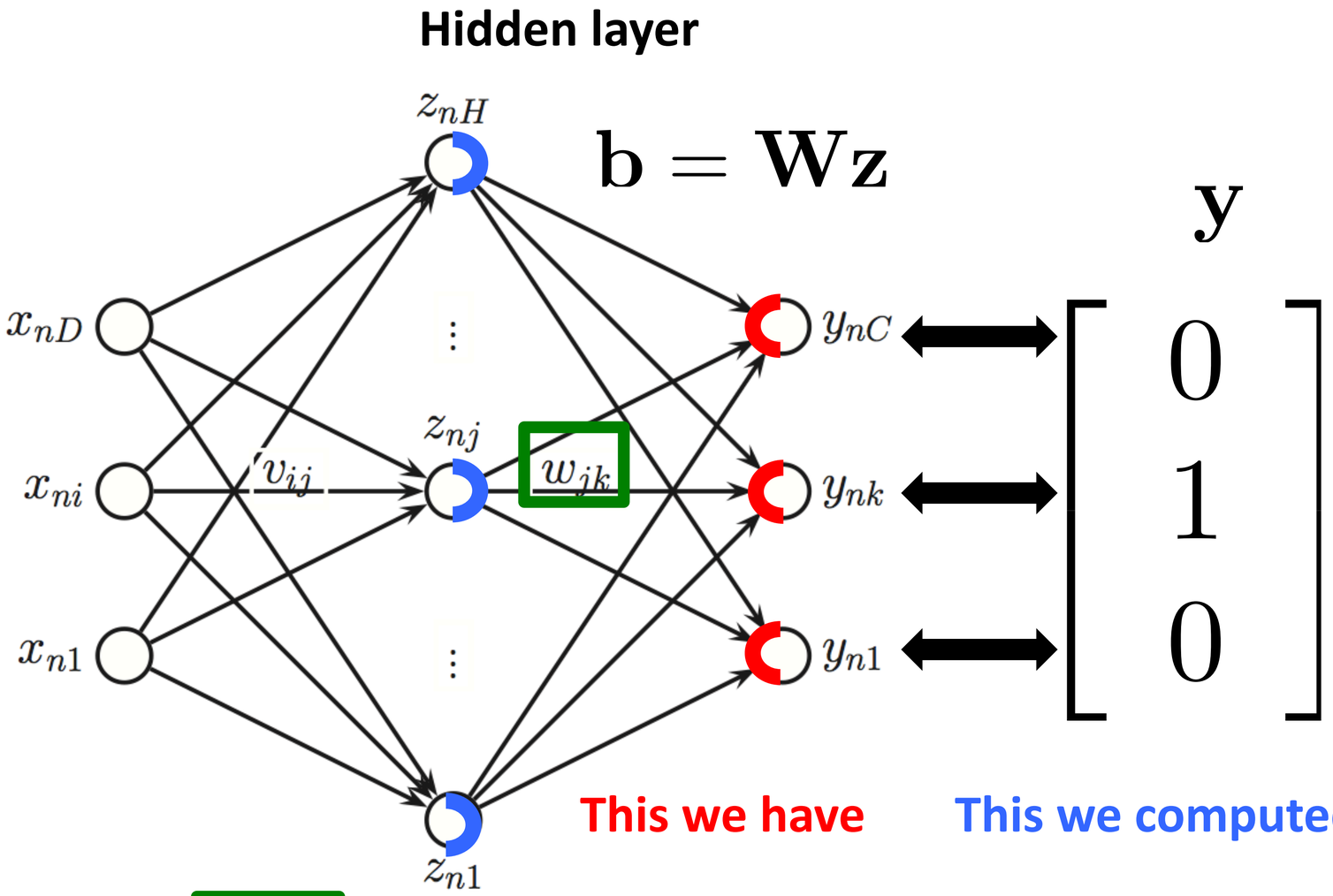
Linear Layer Parameters in Backward: 1-to-1

$$b_m = \sum_{h=1}^{H} z_h w_{h,m}$$



$$\frac{\partial L}{\partial w_{h,m}} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial w_{h,m}} = \frac{\partial L}{\partial b_m} z_h$$

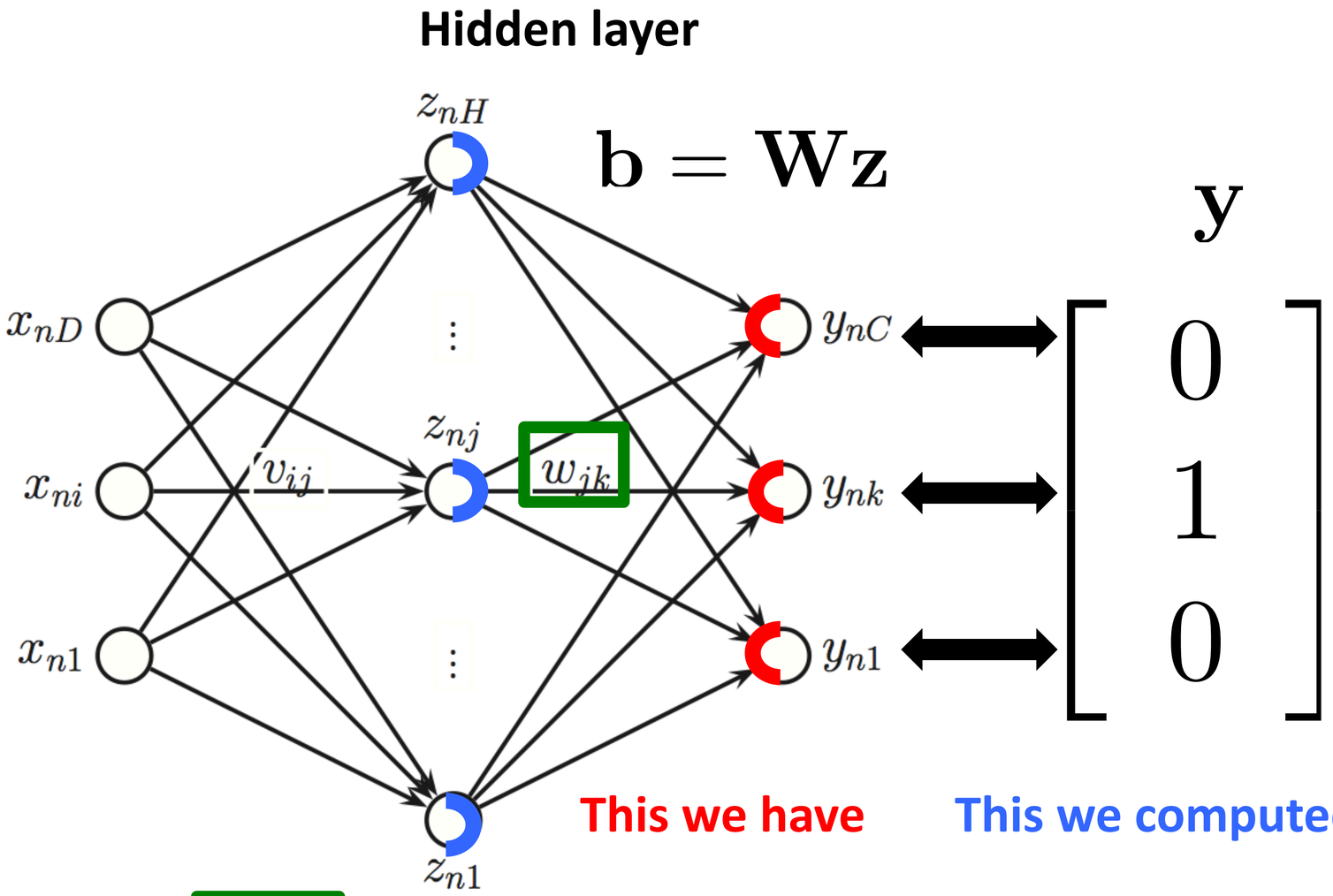
A Neural Network in Backward Mode



This we want

$$\frac{\partial l}{\partial w_{jk}} = \sum_m \frac{\partial l}{\partial b_m} \frac{\partial b_m}{\partial w_{jk}}$$

A Neural Network in Backward Mode



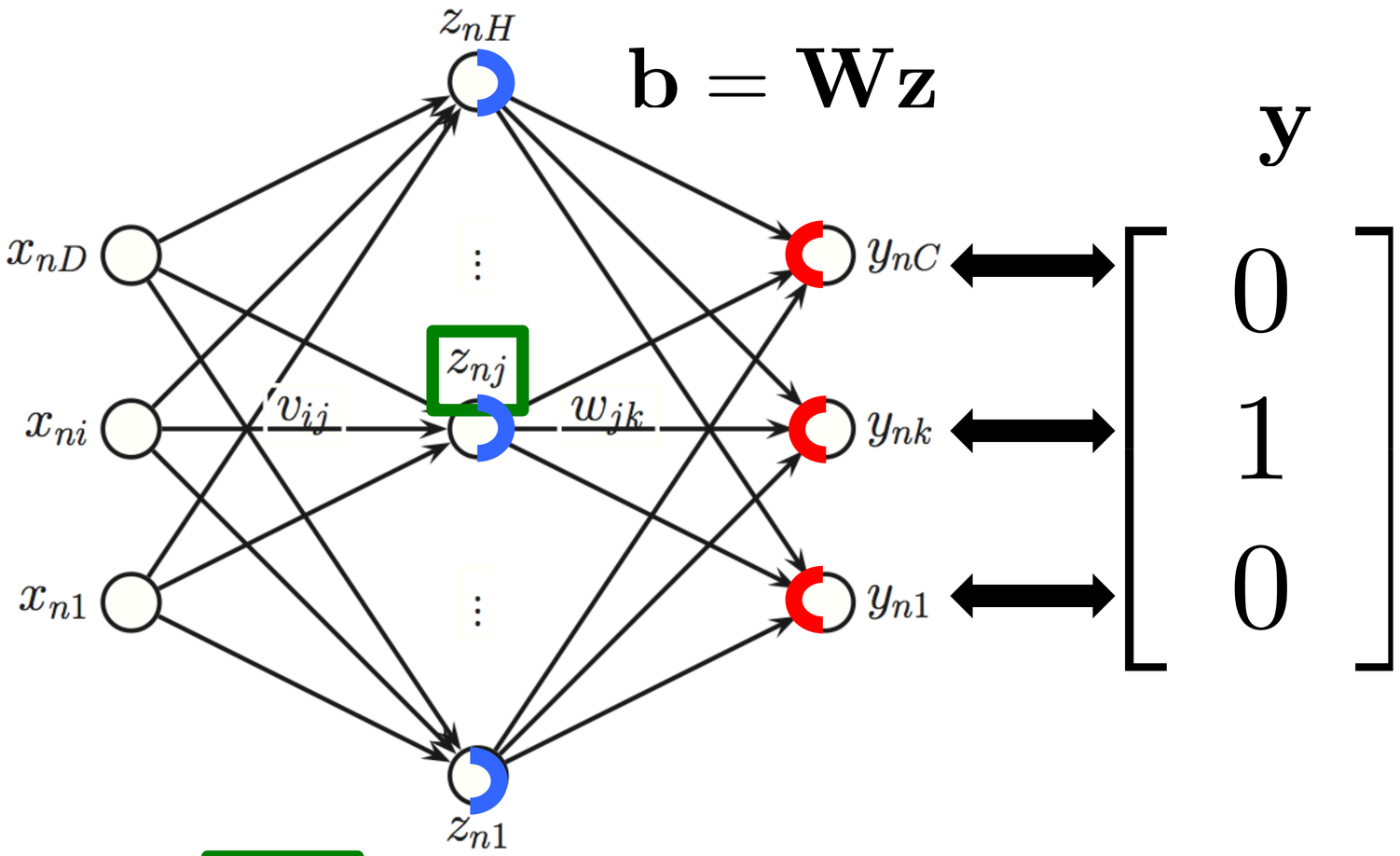
This we have

This we computed

This we want

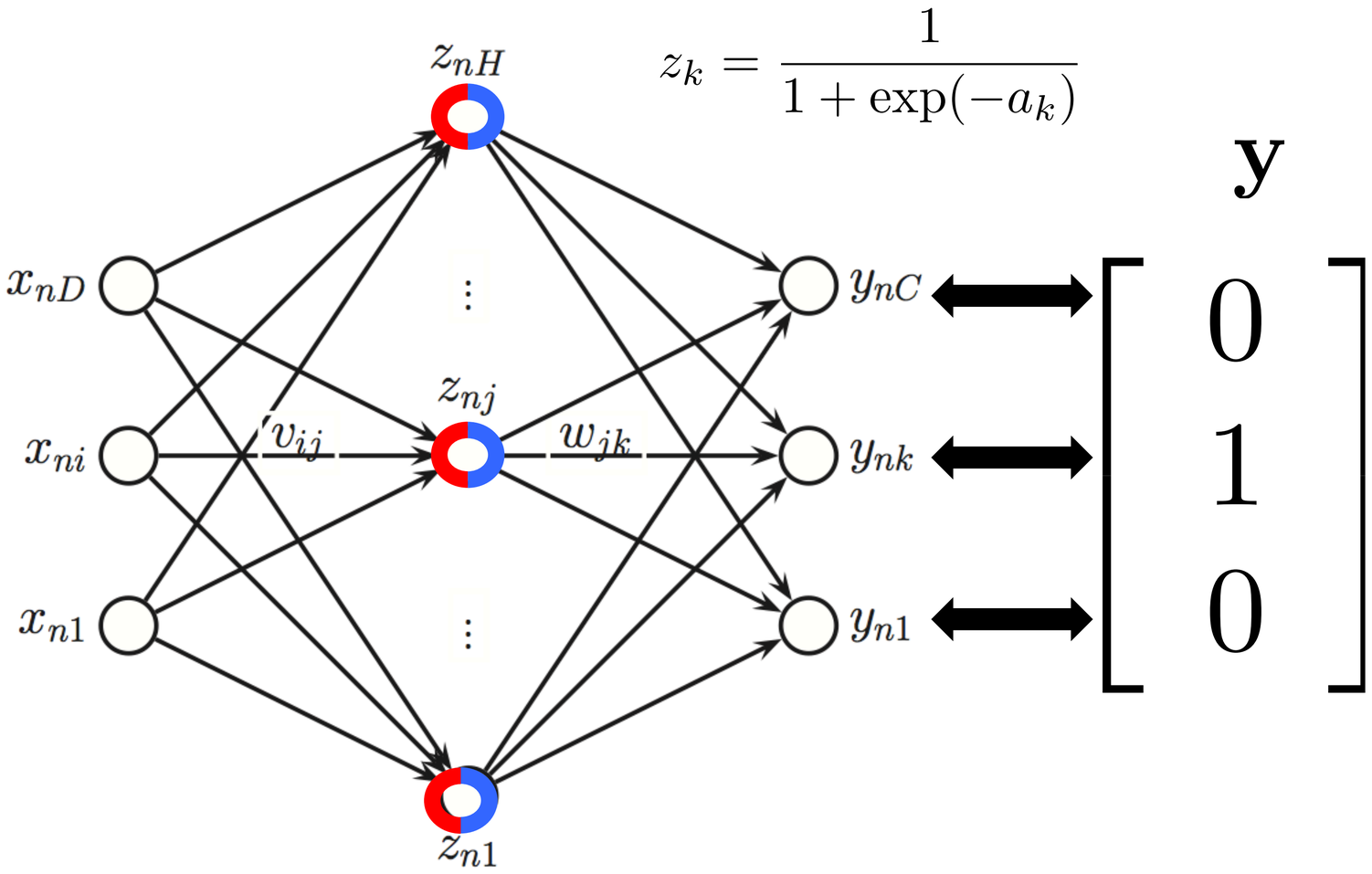
$$\frac{\partial l}{\partial w_{jk}} = \sum_m \frac{\partial l}{\partial b_m} \frac{\partial b_m}{\partial w_{jk}} = \frac{\partial l}{\partial b_m} z_j$$

A Neural Network in Backward Mode

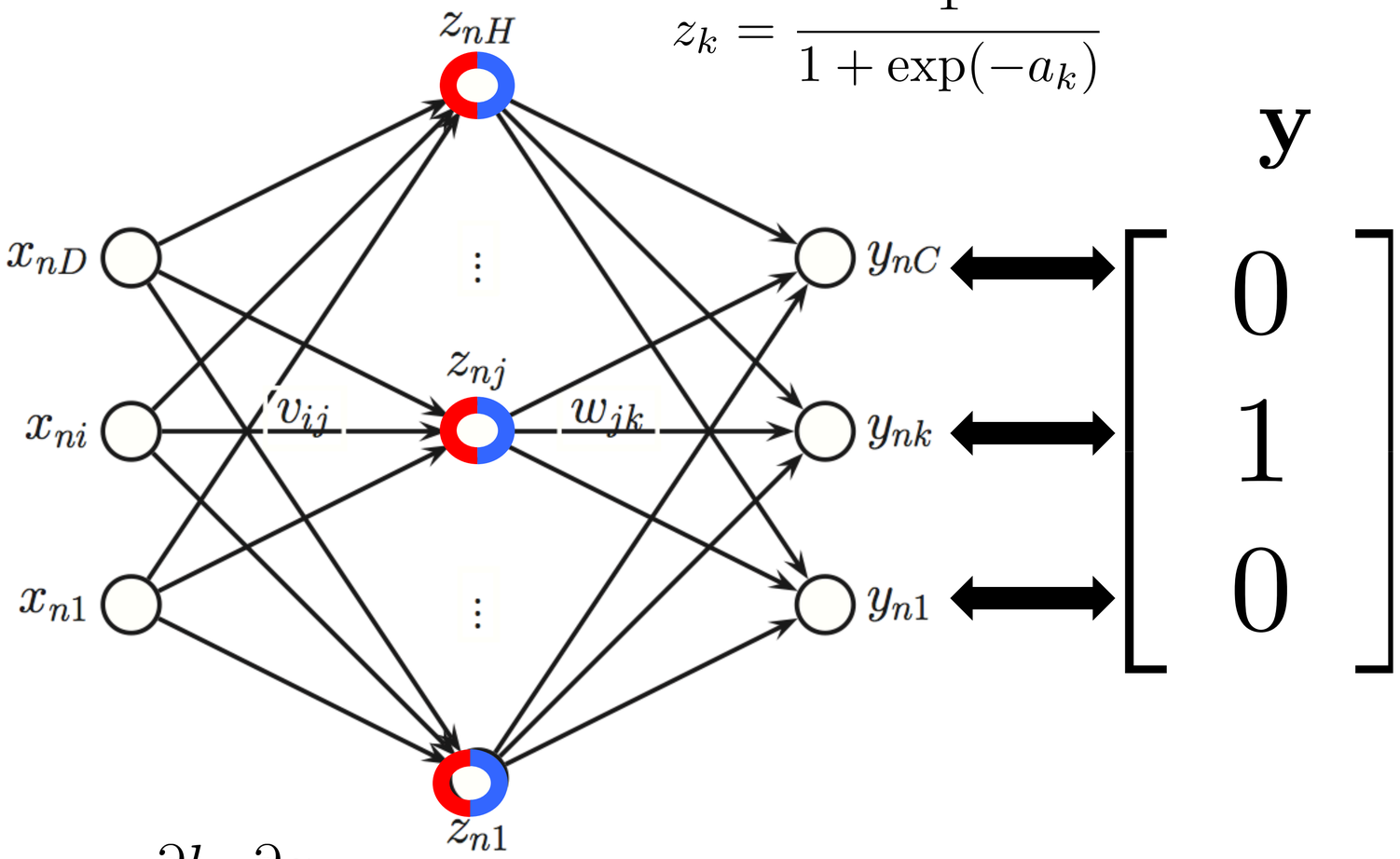


$$\boxed{\frac{\partial l}{\partial z_j}} = \sum_m \boxed{\frac{\partial l}{\partial b_m}} \boxed{\frac{\partial b_m}{\partial z_j}} = \sum_m \frac{\partial l}{\partial b_m} w_{j,m}$$

A Neural Network in Backward Mode

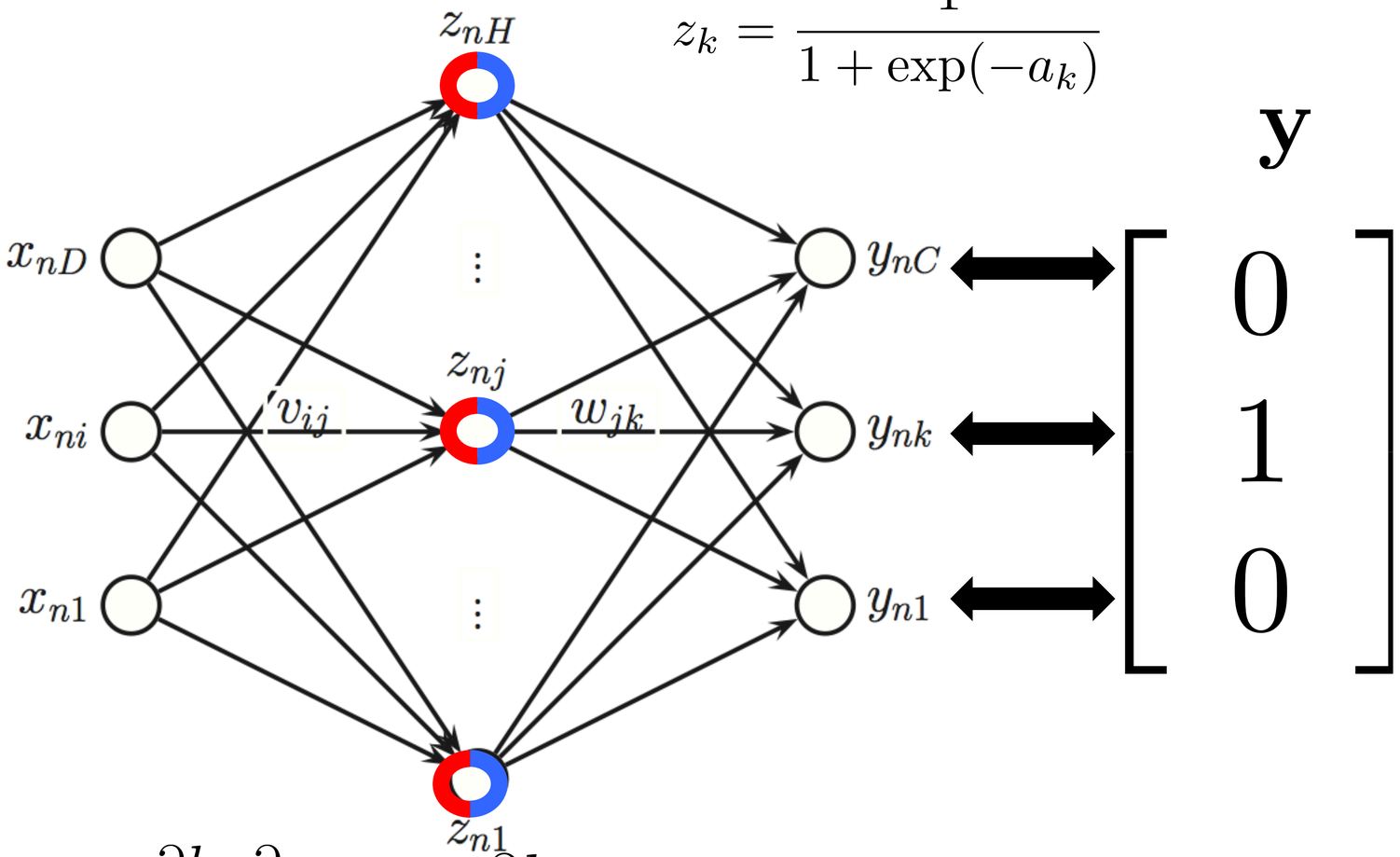


A Neural Network in Backward Mode



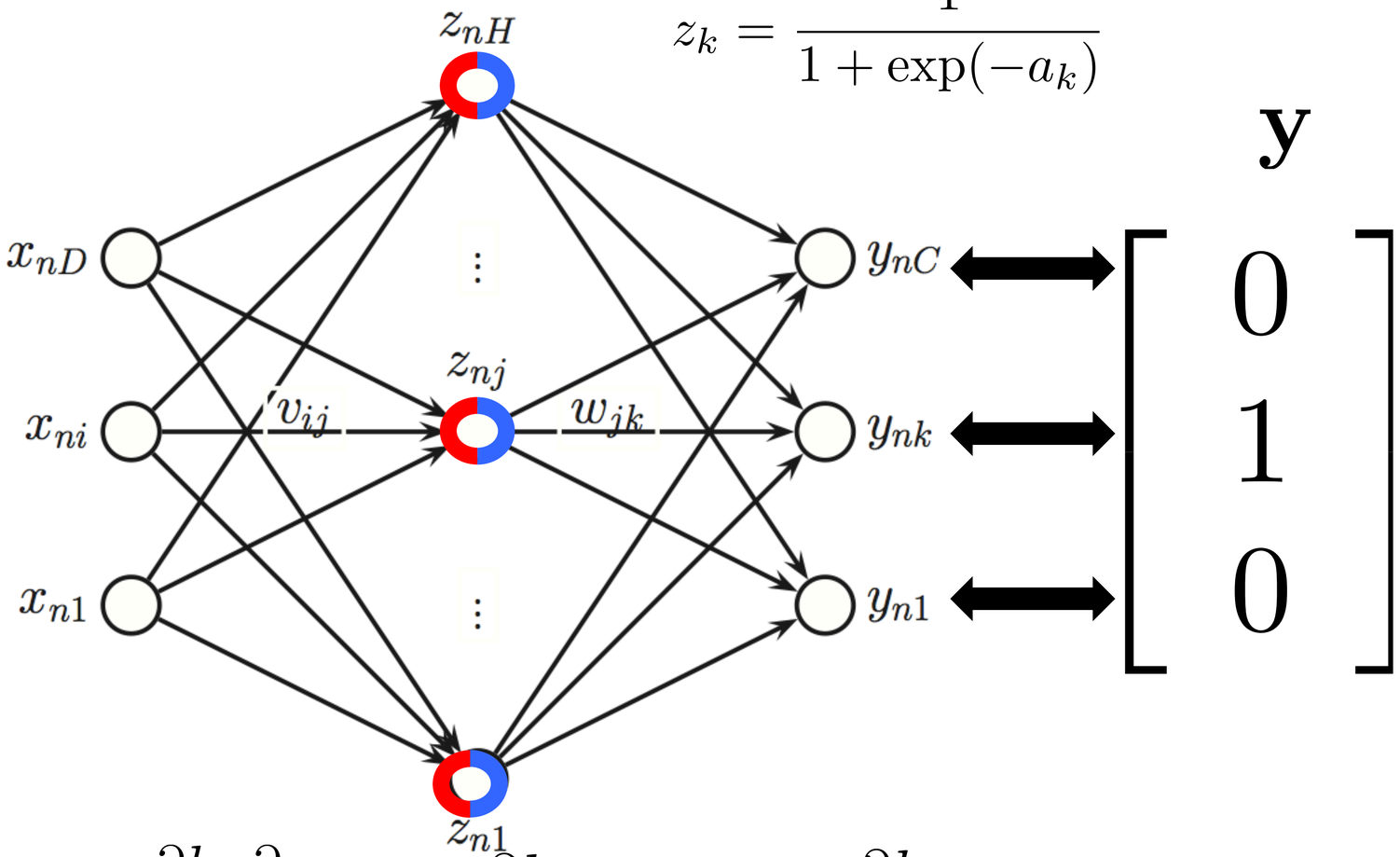
$$\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m} \frac{\partial z_m}{\partial a_k}$$

A Neural Network in Backward Mode



$$\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m}\frac{\partial z_m}{\partial a_k} = \frac{\partial l}{\partial z_k} g'(a_k)$$

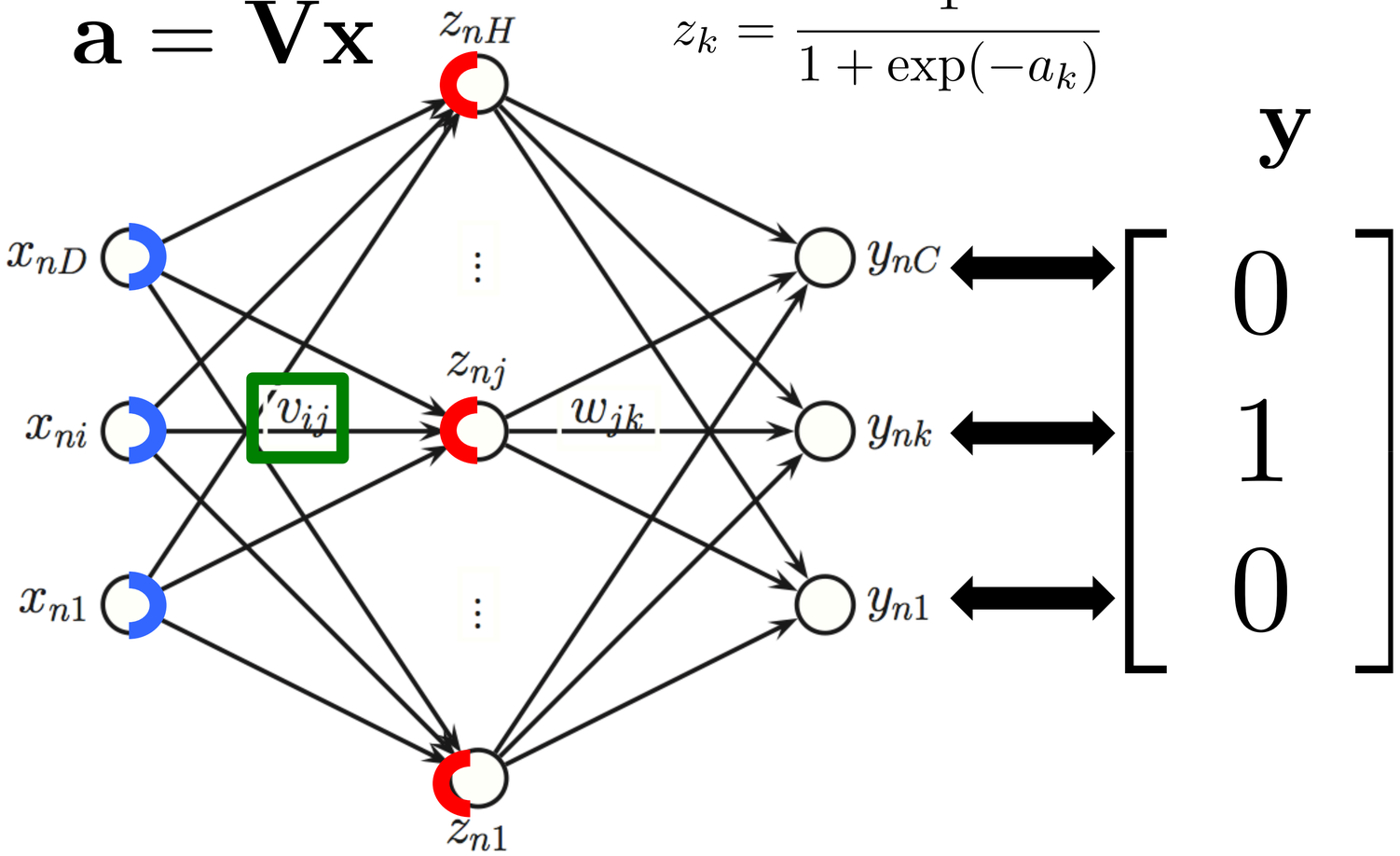
A Neural Network in Backward Mode



$$\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m}\frac{\partial z_m}{\partial a_k} = \frac{\partial l}{\partial z_k} g'(a_k) = \frac{\partial l}{\partial z_k} g(a_k)(1-g(a_k))$$

A Neural Network in Backward Mode

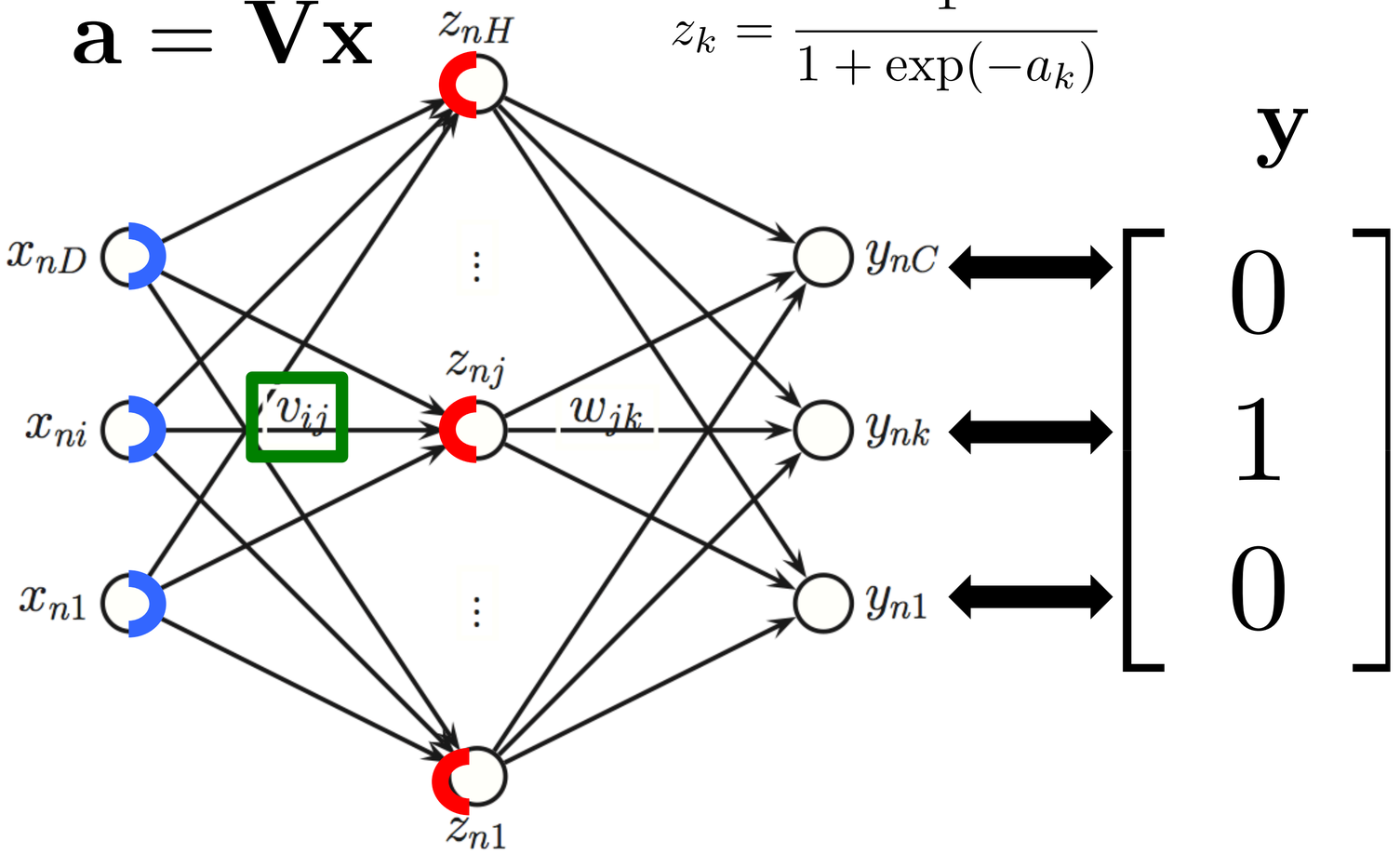
$$\mathbf{a} = \mathbf{V}\mathbf{x} \qquad z_k = \frac{1}{1 + \exp(-a_k)}$$



A Neural Network in Backward Mode

$$\mathbf{a} = \mathbf{V}\mathbf{x}$$

$$z_k = \frac{1}{1 + \exp(-a_k)}$$

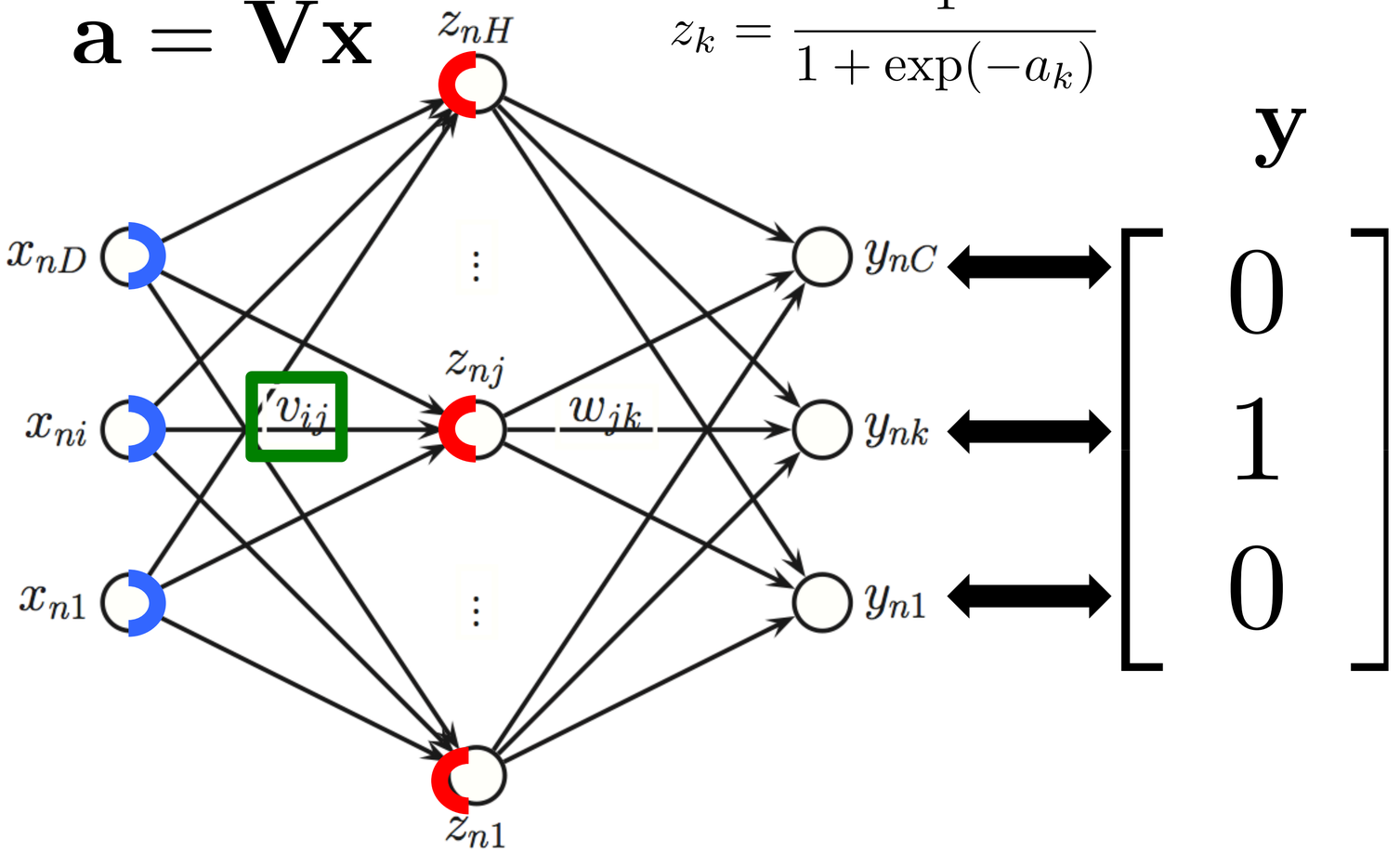


$$\frac{\partial l}{\partial v_{ij}} = \sum_k \frac{\partial l}{\partial a_k} \frac{\partial a_k}{\partial v_{ij}}$$

A Neural Network in Backward Mode

$$\mathbf{a} = \mathbf{V}\mathbf{x}$$

$$z_k = \frac{1}{1 + \exp(-a_k)}$$



$$\frac{\partial l}{\partial v_{ij}} = \sum_k \frac{\partial l}{\partial a_k} \frac{\partial a_k}{\partial v_{ij}} = \frac{\partial l}{\partial a_j} x_i$$

Neural Network Training: Old and New Tricks

- Old
  - Backpropagation algorithm
  - **Stochastic gradient, momentum, weight decay**

- New
  - Dropout
  - Relu
  - Batch Norm(alization), GroupNorm, Spectral Normalization
  - Res(idual) Net(work)

Training Objective for N training samples

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \sum_l \lambda_l \sum_{k,m} (\mathbf{W}_{k,m}^l)^2$$

Per-sample loss

Per-layer regularization

Training Objective for N training samples

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \sum_l \lambda_l \sum_{k,m} (\mathbf{W}_{k,m}^l)^2$$

Per-sample loss

Per-layer regularization

Gradient descent: $\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}_t)$

Training Objective for N training samples

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \sum_l \lambda_l \sum_{k,m} (\mathbf{W}_{k,m}^l)^2$$

Per-sample loss

Per-layer regularization

Gradient descent: $\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}_t)$

(l,k,m) element of gradient vector:

Training Objective for N training samples

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \sum_l \lambda_l \sum_{k,m} (\mathbf{W}_{k,m}^l)^2$$

Per-sample loss

Per-layer regularization

Gradient descent: $\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}_t)$

(l,k,m) element of gradient vector:

$$\frac{\partial L}{\partial \mathbf{W}_{k,m}^l} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}_{k,m}^l} + 2\lambda_l \mathbf{W}_{k,m}^l$$

Training Objective for N training samples

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \sum_l \lambda_l \sum_{k,m} (\mathbf{W}_{k,m}^l)^2$$

Per-sample loss

Per-layer regularization

Gradient descent: $\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}_t)$

(l,k,m) element of gradient vector:

$$\frac{\partial L}{\partial \mathbf{W}_{k,m}^l} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}_{k,m}^l} + 2\lambda_l \mathbf{W}_{k,m}^l$$

Back-prop for i-th example

Training Objective for N training samples

$$L(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \sum_l \lambda_l \sum_{k,m} (\mathbf{W}_{k,m}^l)^2$$

Per-sample loss Per-layer regularization

Gradient descent: $\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}_t)$

(l,k,m) element of gradient vector:

$$\frac{\partial L}{\partial \mathbf{W}_{k,m}^l} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}_{k,m}^l} + 2\lambda_l \mathbf{W}_{k,m}^l$$

Back-prop for i-th example

If N=10^6 , to update **W** **once** need to back-prop 10^6 times!

Regularization in SGD: Weight Decay

Gradient: **Batch:** [1..N]

$$\frac{\partial L}{\partial \mathbf{W}_{k,m}^{l}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}_{k,m}^{l}} + 2\lambda_l \mathbf{W}_{k,m}^{l}$$

Regularization in SGD: Weight Decay

Gradient: **Batch:** [1..N]

$$\frac{\partial L}{\partial \mathbf{W}^l_{k,m}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}^l_{k,m}} + 2\lambda_l \mathbf{W}^l_{k,m}$$

Noisy **('Stochastic') Gradient**:

Minibatch: B elements

b(1), b(2),…, b(B): ***randomly*** sampled from [1,N]

Regularization in SGD: Weight Decay

Gradient: **Batch:** [1..N]

$$\frac{\partial L}{\partial \mathbf{W}_{k,m}^{l}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}_{k,m}^{l}} + 2\lambda_l \mathbf{W}_{k,m}^{l}$$

Noisy **('Stochastic') Gradient**:

Minibatch: B elements

b(1), b(2),…, b(B): ***randomly*** sampled from [1,N]

$$\frac{\partial L}{\partial \mathbf{W}_{k,m}^{l}} \simeq \frac{1}{B}\sum_{i=1}^{B}\frac{\partial l(\mathbf{y}^{b(i)}, \hat{\mathbf{y}}^{b(i)})}{\partial \mathbf{W}_{k,m}^{l}} + 2\lambda_l \mathbf{W}_{k,m}^{l}$$

Back-prop on minibatch

Regularization in SGD: Weight Decay

Gradient: **Batch:** [1..N]

$$\frac{\partial L}{\partial \mathbf{W}^l_{k,m}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial l(\mathbf{y}^i, \hat{\mathbf{y}}^i)}{\partial \mathbf{W}^l_{k,m}} + 2\lambda_l \mathbf{W}^l_{k,m}$$

Noisy **('Stochastic') Gradient**:

Minibatch: B elements

b(1), b(2),…, b(B): ***randomly*** sampled from [1,N]

$$\frac{\partial L}{\partial \mathbf{W}^l_{k,m}} \simeq \frac{1}{B}\sum_{i=1}^{B}\frac{\partial l(\mathbf{y}^{b(i)}, \hat{\mathbf{y}}^{b(i)})}{\partial \mathbf{W}^l_{k,m}} + 2\lambda_l \mathbf{W}^l_{k,m}$$

Back-prop on minibatch

Weight decay

Regularization in SGD: Weight Decay

Gradient: **Batch:** [1..N]

$$\frac{\partial L}{\partial \mathbf{W}_{k,m}^{l}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial l(\mathbf{y}^{i},\hat{\mathbf{y}}^{i})}{\partial \mathbf{W}_{k,m}^{l}} + 2\lambda_{l}\mathbf{W}_{k,m}^{l}$$

Noisy **('Stochastic') Gradient**:

Minibatch: B elements

b(1), b(2),…, b(B): ***randomly*** sampled from [1,N]

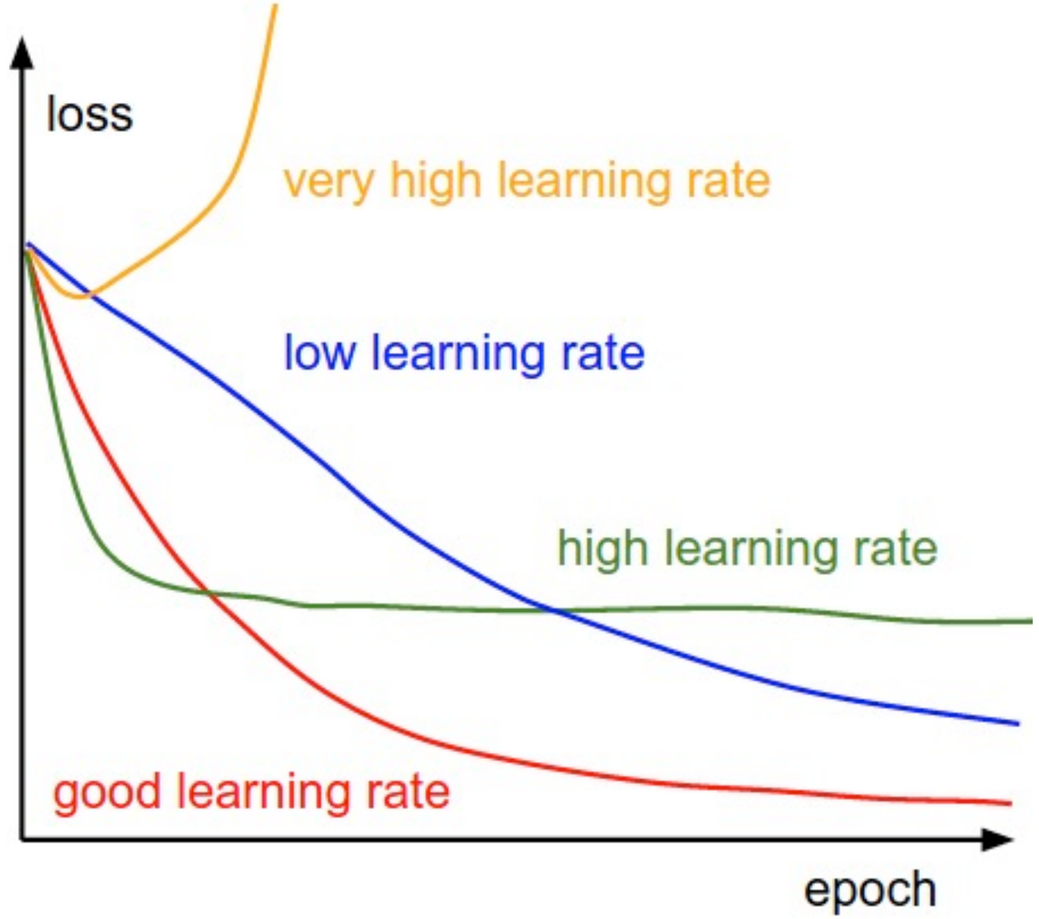
$$\frac{\partial L}{\partial \mathbf{W}_{k,m}^{l}} \simeq \frac{1}{B}\sum_{i=1}^{B}\frac{\partial l(\mathbf{y}^{b(i)},\hat{\mathbf{y}}^{b(i)})}{\partial \mathbf{W}_{k,m}^{l}} + 2\lambda_{l}\mathbf{W}_{k,m}^{l}$$

Back-prop on minibatch

Weight decay

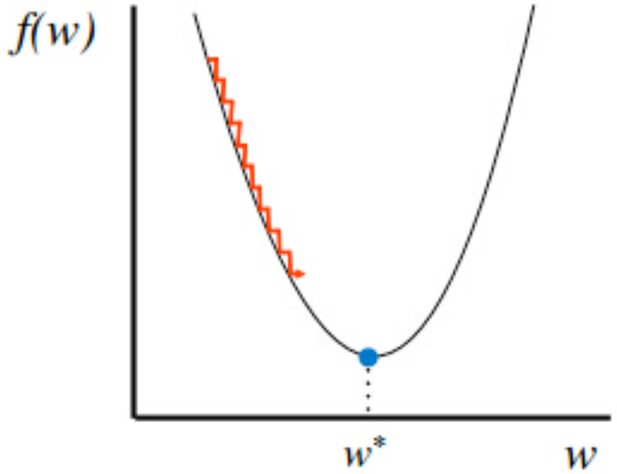
Epoch: N samples, N/B batches

Learning Rate

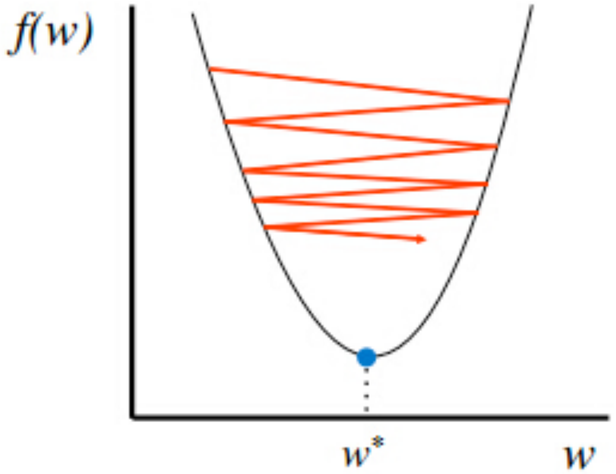


$$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon \nabla_{\mathbf{W}} L(\mathbf{W}_t)$$

(S)GD with Adaptable Stepsize

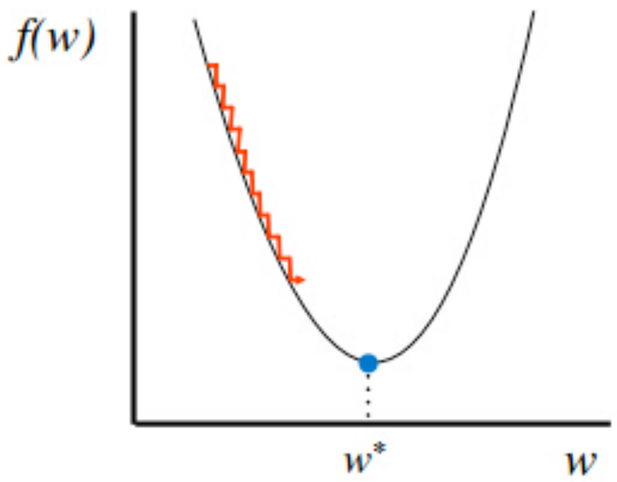


Too small: converge very slowly

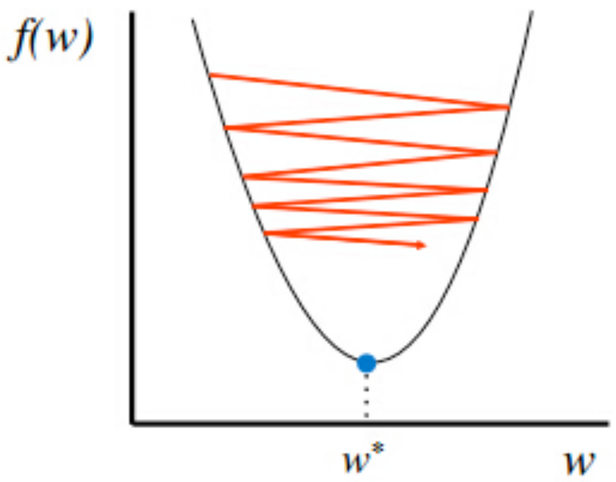


Too big: overshoot and even diverge

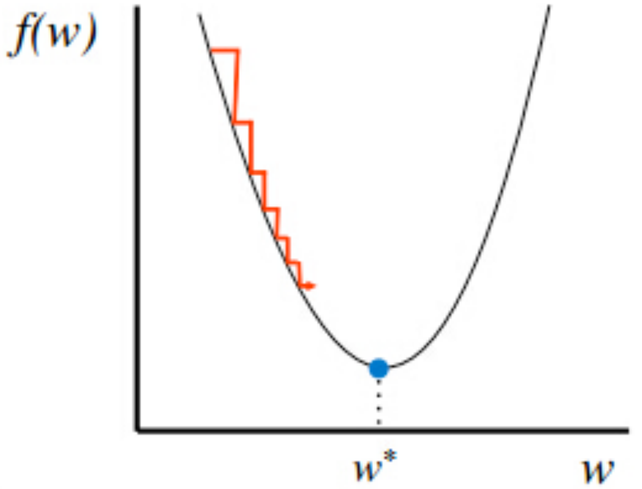
(S)GD with Adaptable Stepsize



Too small: converge very slowly

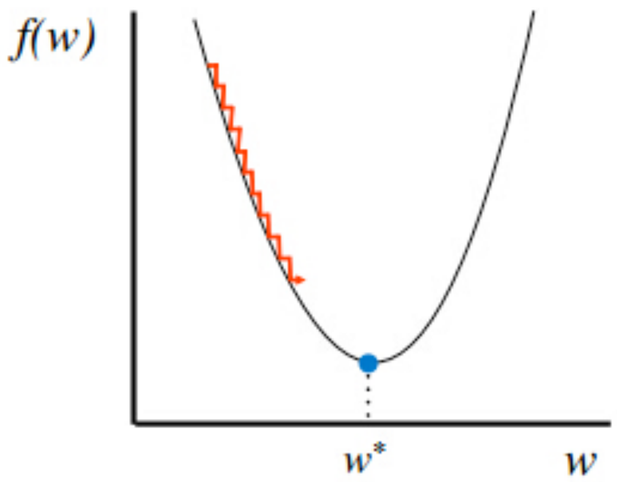


Too big: overshoot and even diverge

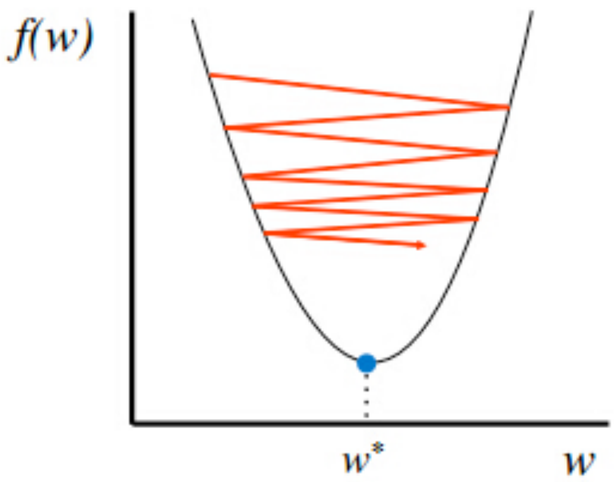


Reduce size over time

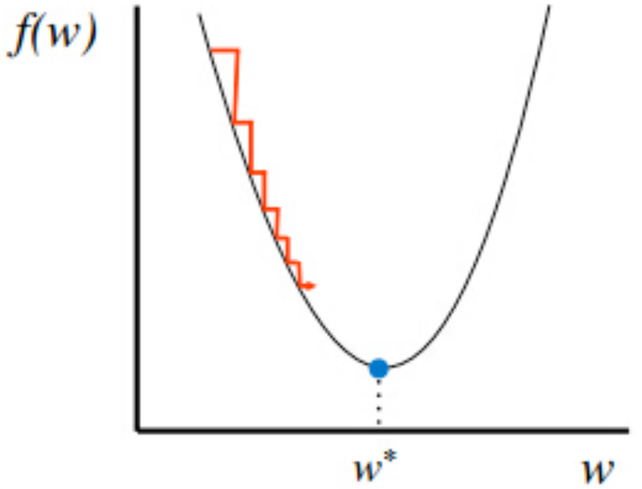
(S)GD with Adaptable Stepsize



Too small: converge very slowly



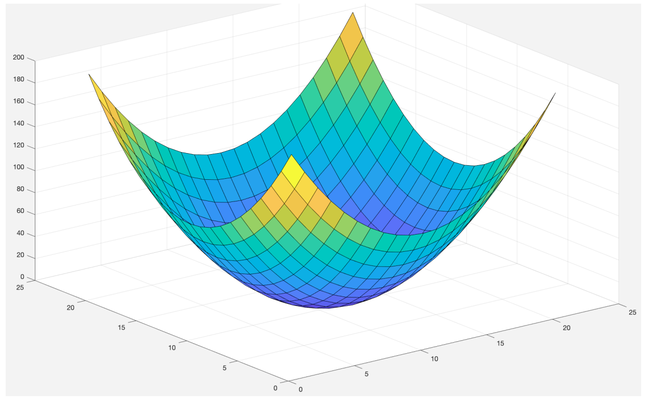
Too big: overshoot and even diverge

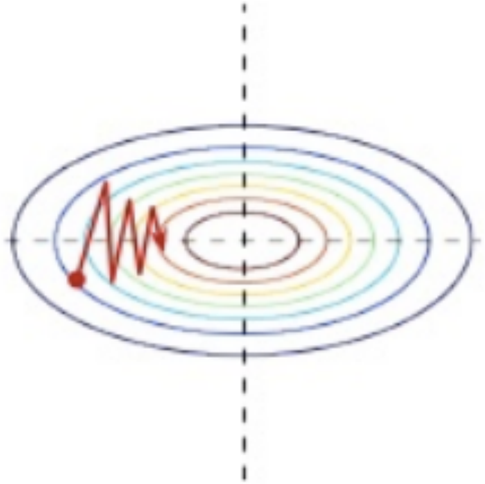


Reduce size over time

e.g. $\epsilon_t = \frac{c}{t}$

(S)GD with Momentum

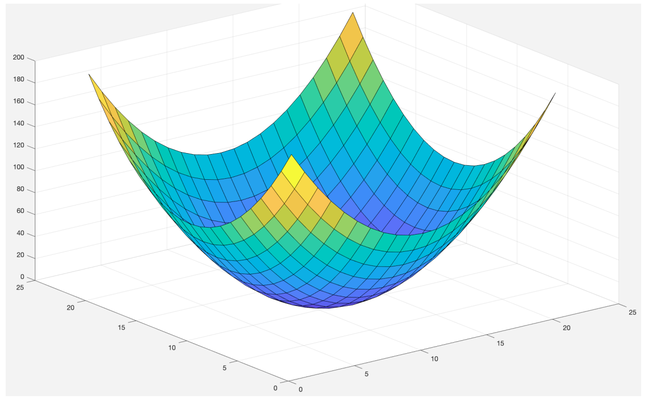


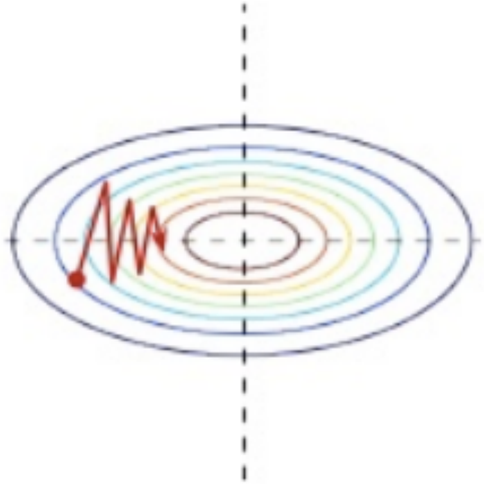


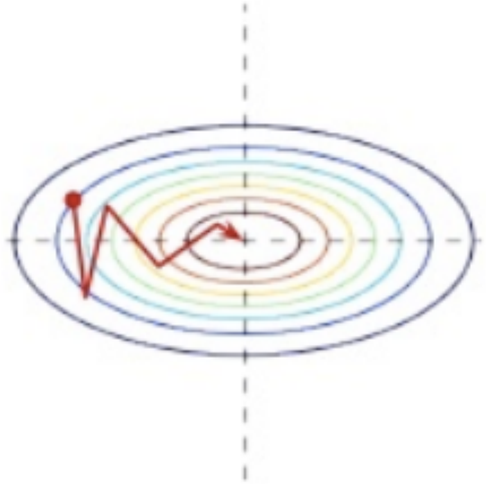
Main idea: retain long-term trend of updates, drop oscillations

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon_t \nabla_{\mathbf{W}} L(\mathbf{W}_t)$$

(S)GD with Momentum



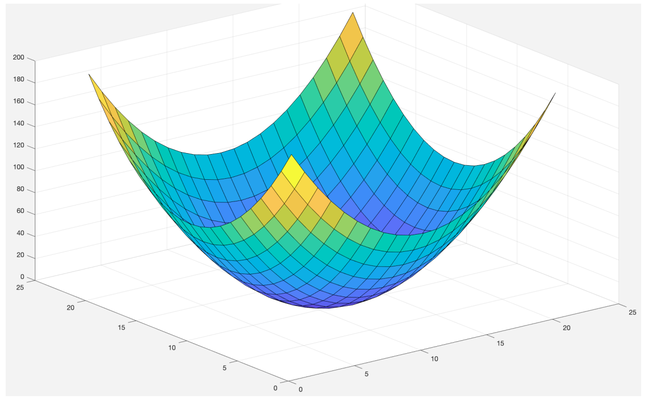


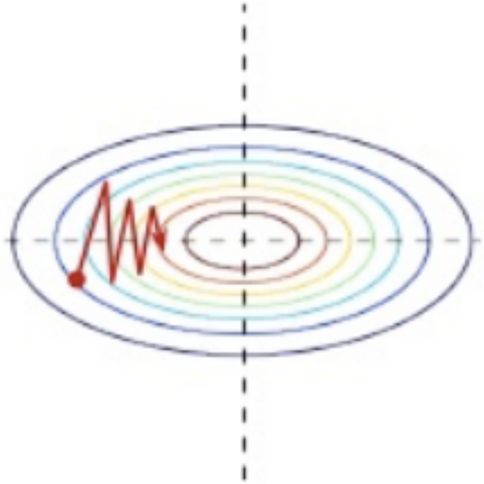


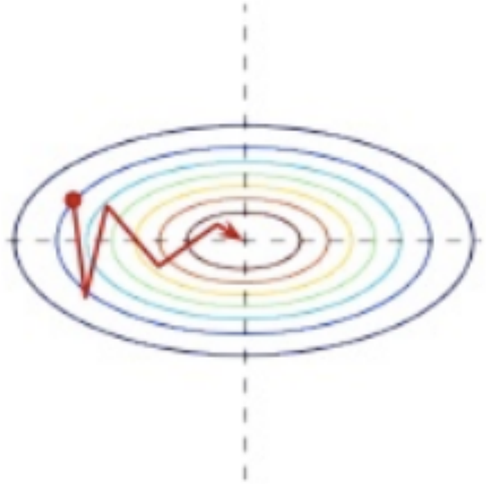
Main idea: retain long-term trend of updates, drop oscillations

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon_t \nabla_{\mathbf{W}} L(\mathbf{W}_t)$$

(S)GD with Momentum



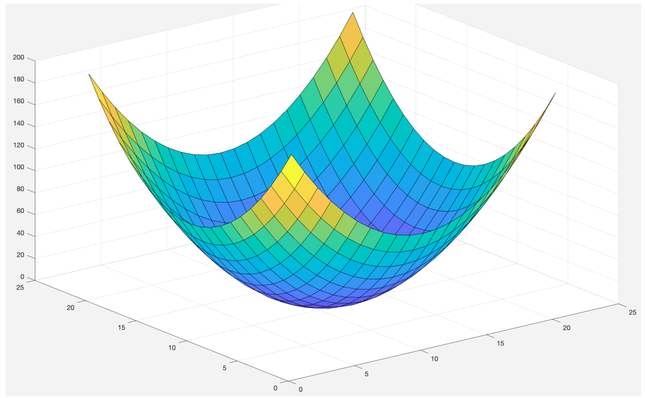


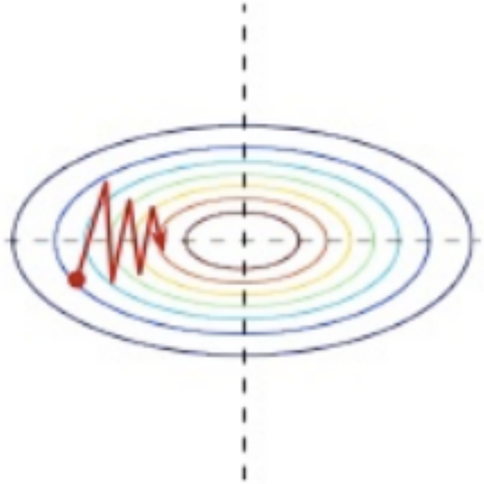


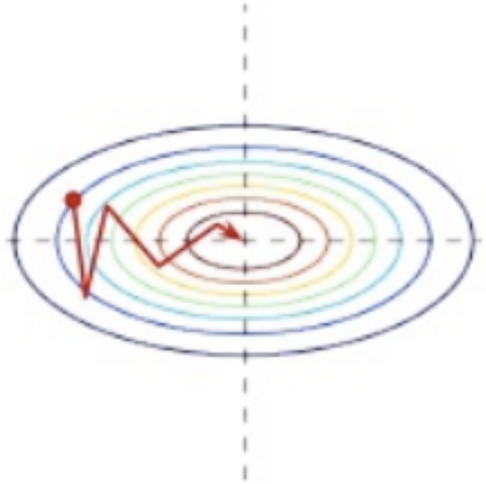
Main idea: retain long-term trend of updates, drop oscillations

(S)GD $$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon_t \nabla_{\mathbf{W}} L(\mathbf{W}_t)$$

(S)GD with Momentum







Main idea: retain long-term trend of updates, drop oscillations

(S)GD $$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon_t \nabla_{\mathbf{W}} L(\mathbf{W}_t)$$

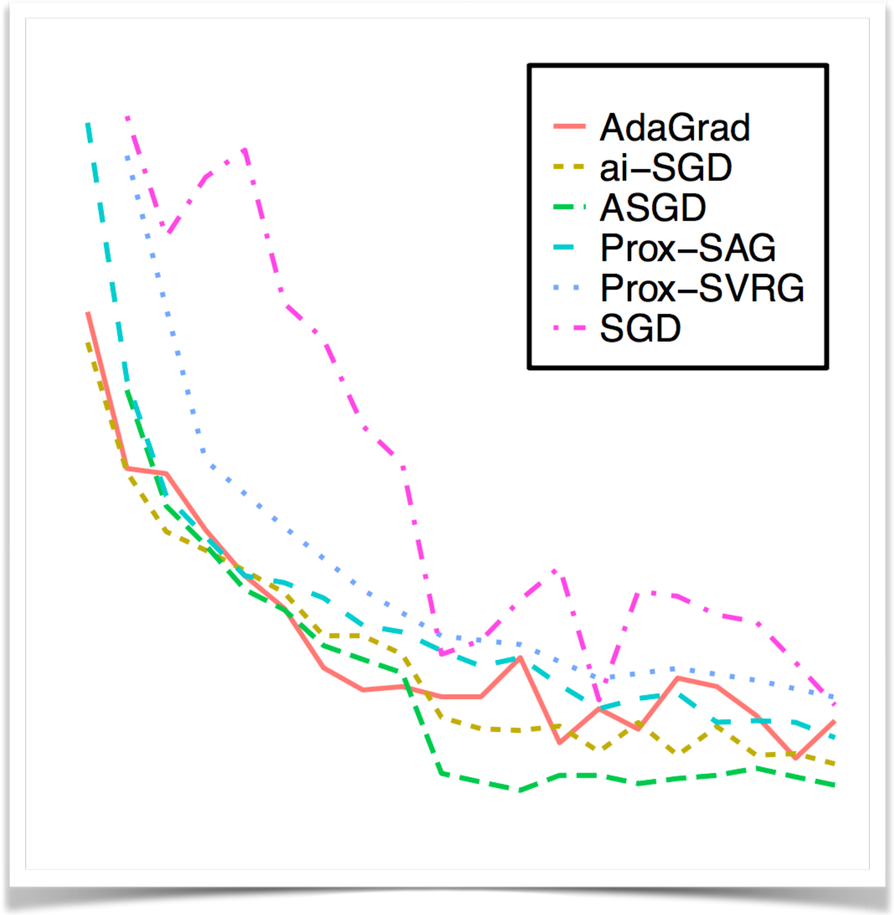
(S)GD + momentum

$$\mathbf{V}_{t+1} = \mu \mathbf{V}_t + (1-\mu) \nabla_{\mathbf{W}} L(\mathbf{W}_t)$$

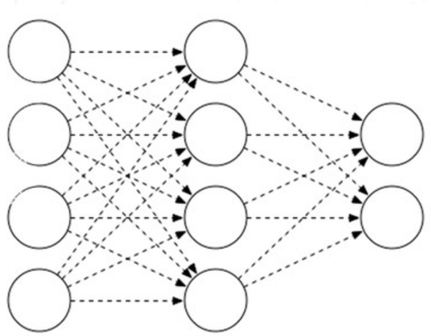
$$\mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon_t \mathbf{V}_{t+1}$$

Step-size Selection & Optimizers

- Nesterov's Accelerated Gradient (NAG)
- R-prop
- AdaGrad
- RMSProp
- AdaDelta
- **Adam**
- …



Course Information (slides/code/comments)



http://geometry.cs.ucl.ac.uk/dl_for_CG/

Scan me