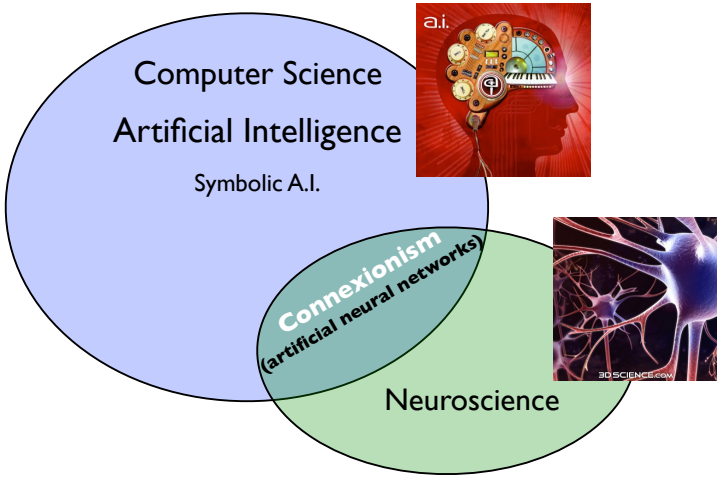
IFT3395/6390 (Prof. Pascal Vincent)

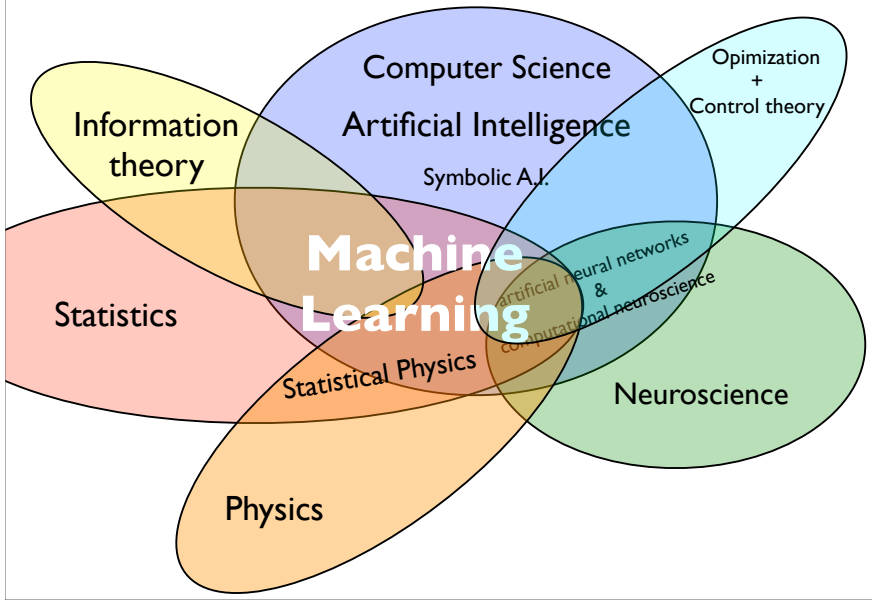
Machine Learning from linear regression to Neural Networks

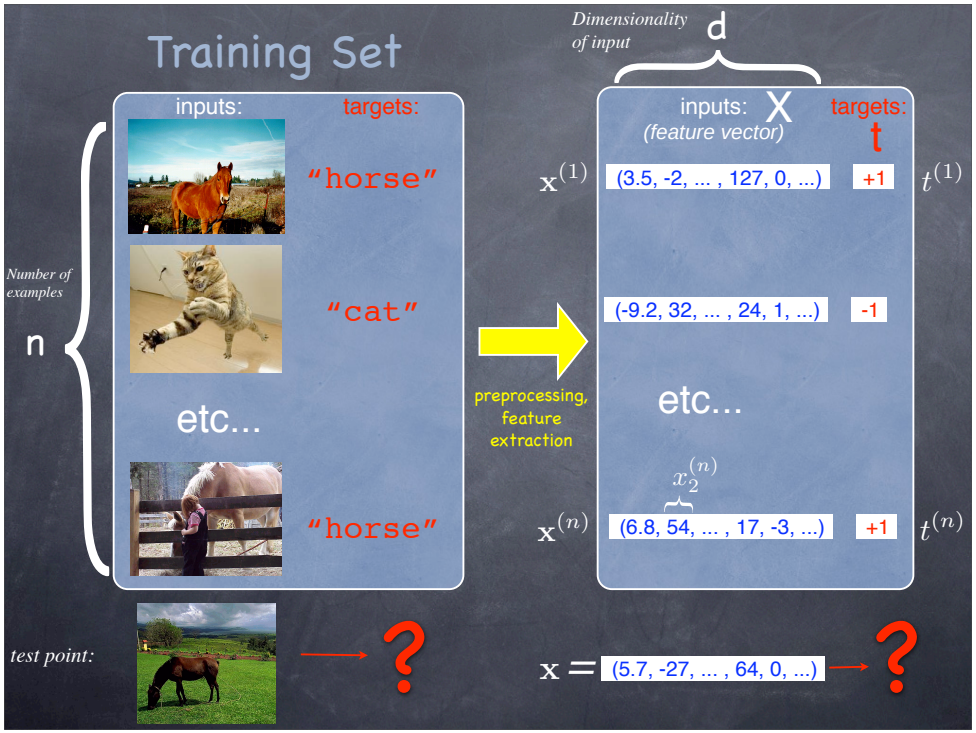
- Introduce machine-learning and neural networks (terminology)
- Start with simple statistical models
- Feed Forward Neural Networks
  (specifically Multilayer Perceptrons)

Historical perspective: **back to 1957**
(Rosenblatt, "Perceptron")



Nowadays vision of the founding disciplines





Machine learning tasks

Supervised learning = predict target t from input x

- t represents a category or "class"
  ➠ classification (binary or multiclass)
- t is a real value
  ➠ regression

Unsupervised learning: no explicit target t

- model the distribution of x
  ➠ density estimation
- capture underlying structure in x
  ➠ dimensionality reduction, clustering, etc...

The task

predicting t from x

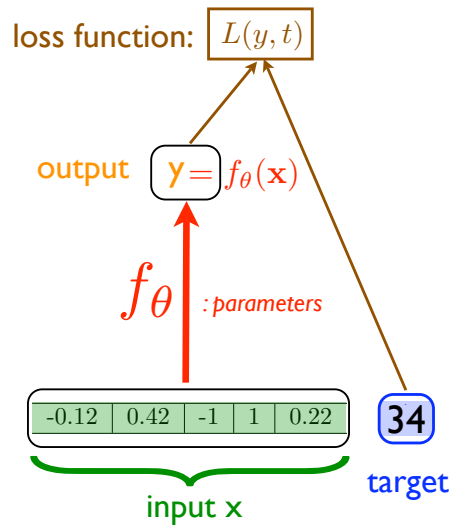
input $\mathbf{x} \in \mathbb{R}^d$ | target t

n examples

$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	t
0.32	-0.27	+1	0	0.82	113
-0.12	0.42	-1	1	0.22	34
0.06	0.35	-1	1	-0.37	56
0.91	-0.72	+1	0	-0.63	77
...	...	...	...	...	...

Training Set D_n

Learning a parameterized function f_θ that minimizes a loss.



Empirical risk minimization

We need to specify:

- A form for parameterized function f_θ
- A specific loss function $L(y,t)$

We then define the empirical risk as:

$$\hat{R}(f_\theta, D_n) = \sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), t^{(i)})$$

i.e. overall loss over the training set

Learning amounts to finding optimal parameters:

$$\theta^\star = \arg\min_\theta \hat{R}(f_\theta, D_n)$$

Linear Regression

A simple learning algorithm

We choose

A **linear** mapping:

$$f_\theta(\mathbf{x}) = \underbrace{\langle \mathbf{w}, \mathbf{x} \rangle}_{\text{dot product}} + b \quad \text{with parameters: } \theta = \{\mathbf{w}, b\}, \underbrace{\mathbf{w}}_{\text{weight vector}} \in \mathbb{R}^d, \underbrace{b}_{\text{bias}} \in \mathbb{R}$$

Squared error loss:

$$L(y,t) = (y-t)^2$$

We search the parameters that minimize the overall loss over the training set

$$\theta^\star = \arg\min_\theta \hat{R}(f_\theta, D_n)$$

Simple linear algebra yields an analytical solution.

Linear Regression

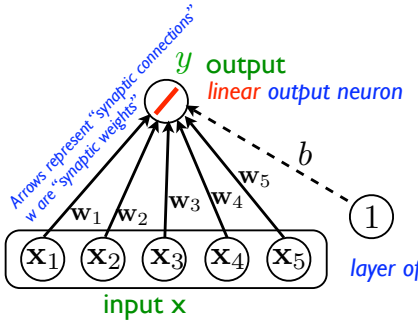
Neural network view

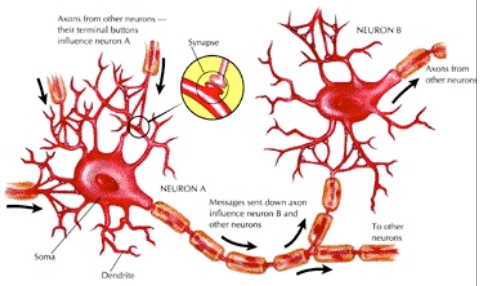
Inuitive understanding of the dot product:

each component of x weighs differently on the response.

$$y = f_\theta(\mathbf{x}) = \mathbf{w}_1\mathbf{x}_1 + \mathbf{w}_2\mathbf{x}_2 + \ldots + \mathbf{w}_d\mathbf{x}_d + b$$

Neural network terminology:





Regularized empirical risk

It may be necessary to induce a preference for some values of the parameters over others to avoid "overfitting"

We can define the regularized empirical risk as:

$$\hat{R}_\lambda(f_\theta, D_n) = \underbrace{\left(\sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), t^{(i)})\right)}_{\text{empirical risk}} + \underbrace{\lambda\Omega(\theta)}_{\text{regularization term}}$$

Ω penalizes more or less certain parameter values

$\lambda \geq 0$ controls the amount of regularization

Ridge Regression

= Linear regression + L2 regularization

We penalize large weights:

$$\Omega(\theta) = \Omega(\mathbf{w}, b) = \|\mathbf{w}\|^2 = \sum_{j=1}^{d} \mathbf{w}_j^2$$

In neural network terminology:

"weight decay" penalty

Again, simple linear algebra yields an analytical solution.

Logistic Regression

If we have a binary classification task: $t \in \{0, 1\}$

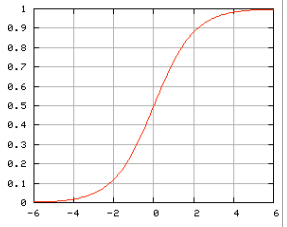
We want to estimate conditional probability: $y \simeq P(t = 1|\mathbf{x})$

$y \in [0, 1]$

We choose

A **non-linear** mapping:

$$f_\theta(\mathbf{x}) = f_{\mathbf{w},b}(\mathbf{x}) = \underbrace{\text{sigmoid}(}_{\text{non-linearity}} \langle \mathbf{w}, \mathbf{x} \rangle + b)$$



$$\textit{logistic}\ \ \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

The *logistic sigmoid* is the inverse of the *logit* "link function" in the terminology of Geleralized Linear Models (GLMs).

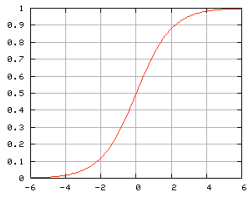
Cross-entropy loss:

$$L(y, t) = t \ln(y) + (1 - t) \ln(1 - y)$$

No analytical solution, but optimization is convex

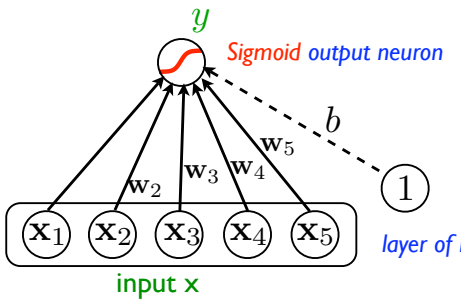
Logistic Regression

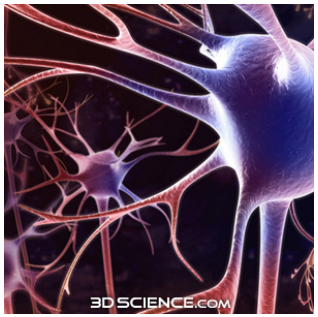
Neural network view



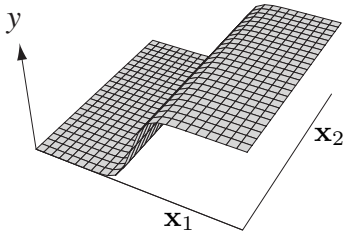
Sigmoid can be viewed as:

- "soft" differentiable alternative to the step function of original Perceptron (Rosenblatt 1957).
- simplified model of "firing rate" response in biological neurons.



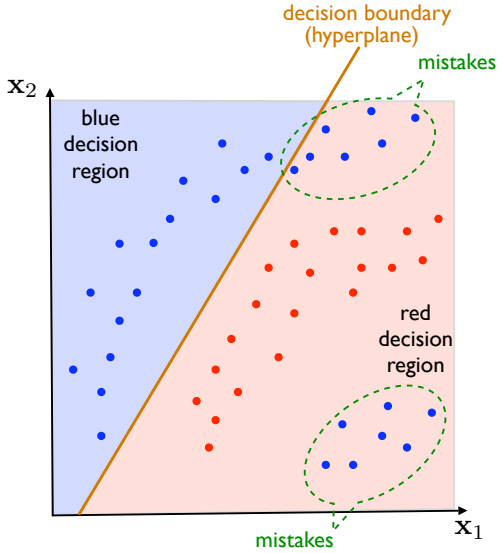


Limitations of Logistic Regression



Only yields "linear" decision boundary: a hyperplane

➠ inappropriate if classes not linearly separable (as on the figure)



How to obtain non-linear decision boundaries ?

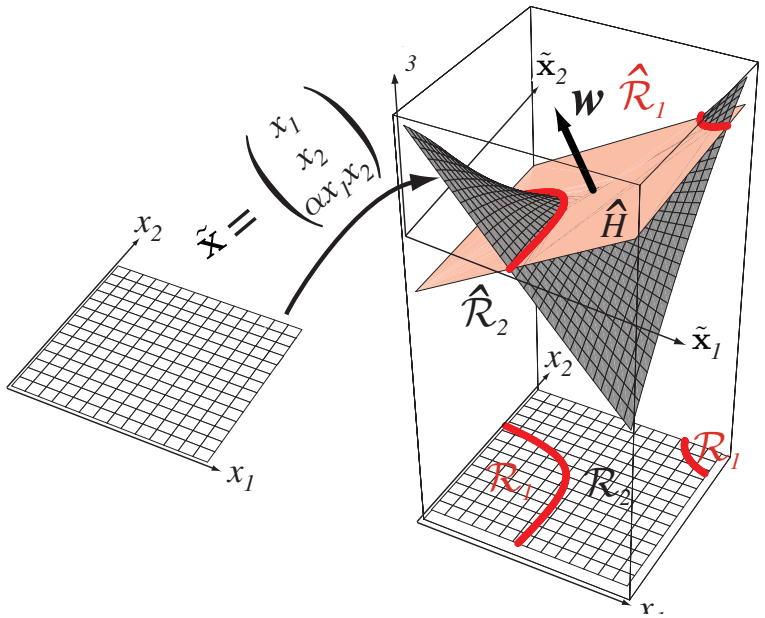
An old technique...

- map x non-linearly to feature space:

$$\tilde{\mathbf{x}} = \phi(\mathbf{x})$$

- find separating hyperplane in new space
- hyperplane in new space corresponds to non-linear decision surface in initial x space.

Ex. using fixed mapping



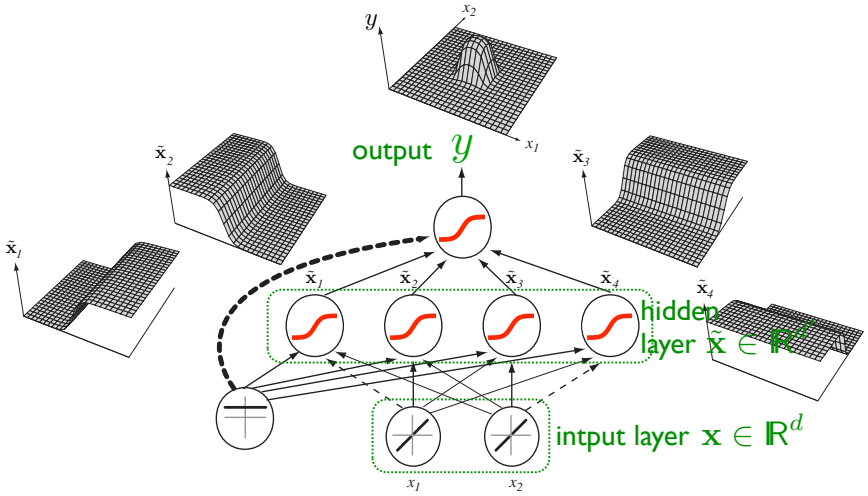
How to obtain non-linear decision boundaries...

Three ways to map x to $\tilde{\mathbf{x}} = \phi(\mathbf{x})$

- Use an explicit fixed mapping
  ➡previous example
- Use an implicit fixed mapping
  ➡Kernel Methods (SVMs, Kernel Logistic Regression ...)
- Learn a parameterized mapping:
  ➡ Multilayer feed-forward Neural Networks
  such as Multilayer Perceptrons (MLP)

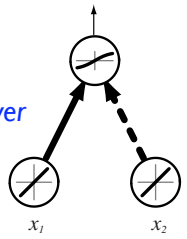
Neural Network:

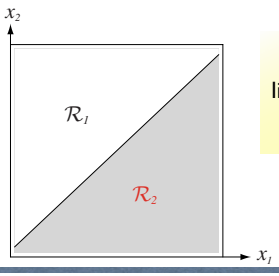
Multi-Layer Perceptron (MLP) with one hidden layer of size 4 neurons



Expressive power of Neural Networks with one hidden layer

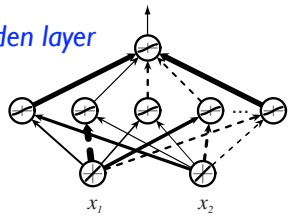
no hidden layer

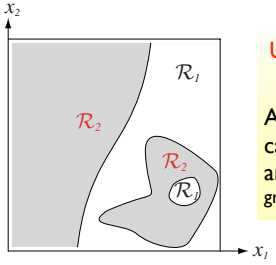




== Logistic regression limited to representing a separating hyperplane

one hidden layer





Universal approximation property

Any continuous function can be approximated arbitarily well (with a growing number of hidden unis)

Neural Network (MLP)

with one hidden layer of size *d'* neurons

Functional form (parametric):

$$y = f_\theta(\mathbf{x}) = \text{sigmoid}\left(\langle \mathbf{w}, \tilde{\mathbf{x}} \rangle + b\right)$$

$$\tilde{\mathbf{x}} = \text{sigmoid}(\underbrace{\mathbf{W}^{\text{hidden}}}_{d' \times d}\mathbf{x} + \underbrace{\mathbf{b}^{\text{hidden}}}_{d' \times 1})$$

Parameters:

$$\theta = \{\mathbf{W}^{\text{hidden}}, \mathbf{b}^{\text{hidden}}, \mathbf{w}, b\}$$

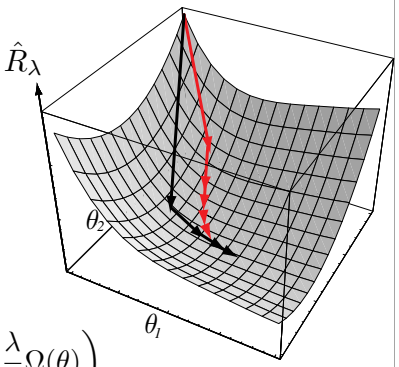
Optimizing parameters on training set *(training the network)*:

$$\theta^\star = \arg\min_\theta \underbrace{\hat{R}_\lambda(f_\theta, D_n)}_{\underbrace{\left(\sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), t^{(i)})\right)}_{\text{empirical risk}} + \underbrace{\lambda\Omega(\theta)}_{\substack{\text{regularization term} \\ \text{(weight decay)}}}}$$

Training Neural Networks

We need to optimize the network's parameters:

$$\theta^\star = \arg\min_\theta \hat{R}_\lambda(f_\theta, D_n)$$



- Initialize parameters at random
- Perform gradient descent

Either batch gradient descent:

REPEAT: $\theta \leftarrow \theta - \eta \frac{\partial \hat{R}_\lambda}{\partial \theta}$

Or stochastic gradient descent:

REPEAT:

Pick i in 1...n

$$\theta \leftarrow \theta - \eta \frac{\partial}{\partial \theta}\left(L(f_\theta(\mathbf{x}^{(i)}), t^{(i)}) + \frac{\lambda}{n}\Omega(\theta)\right)$$

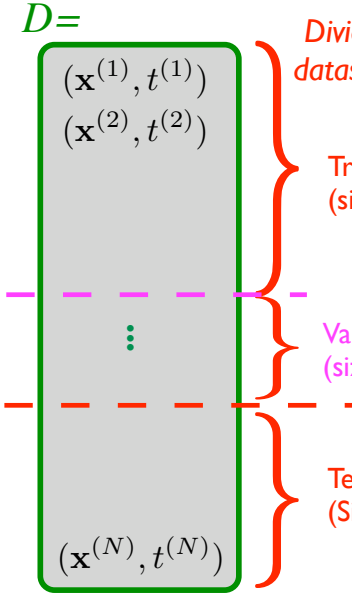
Or other gradient descent technique

(conjugate gradient, Newton, steps natural gradient, ...)

Hyper-parameters controlling capacity

- Network has a set of *parameters*: θ
  - optimized on the training set using gradient descent.
- There are also *hyper-parameters* that control model "capacity"
  - number of hidden units d'
  - regularizaiton control λ (weight decay)
  - early stopping of the optimization
  - tuned by a model selection procedure, **not** on training set.

Hyper-parameter tuning



For each considered values of hyper-parameters:

1) Train the model, i.e. find the value of the parameters that optimize the regularized empirical risk on the training set.

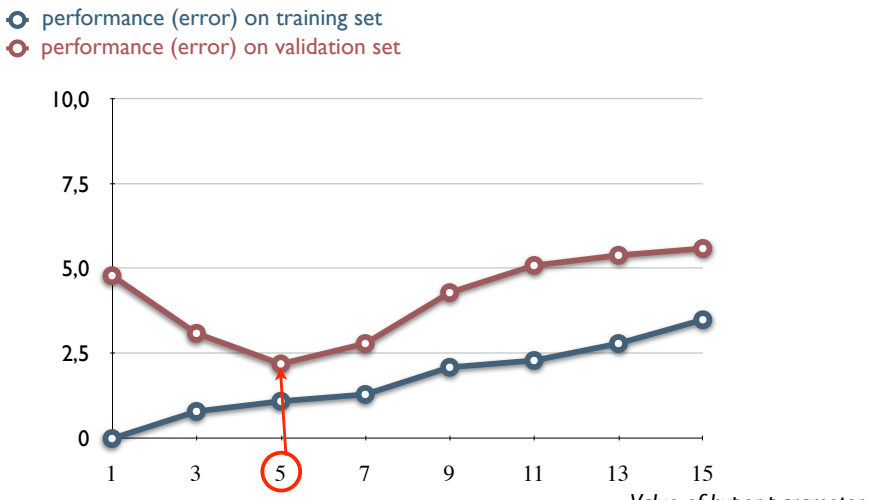
2) Evaluate performance on validation set based on criterion we truly care about.

Keep value of hyper-parameters with best performance on validation set.
(possibly retrain on union of train and validation).

Evaluate generalization performance on separate test-set never used during training or validation (i.e. unbiased "out-of-sample" evaluation).

If too few examples, use k-fold cross-validation or leave-one-out ("jack-knife")

Hyper-parameter tuning



hyper-parameter value yielding smallest error
on validation set is **5** (whereas it's **1** on the training set)

Summary

- Feed-forward Neural Networks (such as Multilayer Perceptrons MLPs) are parameterized non-linear functions or "Generalized non-linear models"...
- ...trained using gradient descent techniques
- Architectural details and capacity-control hyper-parameters must be tuned with proper model selection procedure.
- Data must be preprocessed into suitable format
  standardization for continuous variable: use $\frac{x-\mu}{\sigma}$
  one-hot encoding for categorical variables ex: [0, 0, 1, 0]

Note: there are many other types of Neural Nets...

Neural Networks

Why they matter
for data mining

- advantages of Neural Networks for data-mining.
- motivating research on learning deep networks.

Advantages of Neural Networks

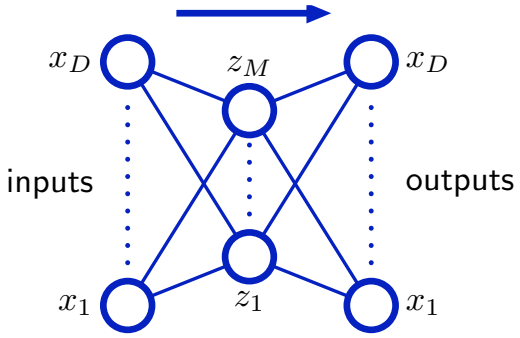
✶The power of learnt non-linearity:
automatically extracting the necessary features

✶Flexibility: they can be used for

- binary classification
- multiclass classification
- regression
- conditional density modeling
  (NNet trained to output parameters of distribution of t as a function of x)
- dimensionality reduction
- ... very adaptable framework (some would say too much...)

Ex: using a Neural Net for dimensionality reduciton

The classical *auto-encoder* framework
learning a lower-dimensional representation



Advantages of Neural Networks
(continued)

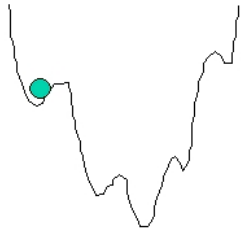
✴Neural Networks **scale well**

- Data-mining often deals with huge databases
- Stochastic gradient descent can handle these
- Many more modern machine-learning techniques have big scaling issues
  (e.g. SVMs and other Kernel methods)

Why then have they gone out of fashion in machine learning ?

- Tricky to train (many hyper-parameters to tune)

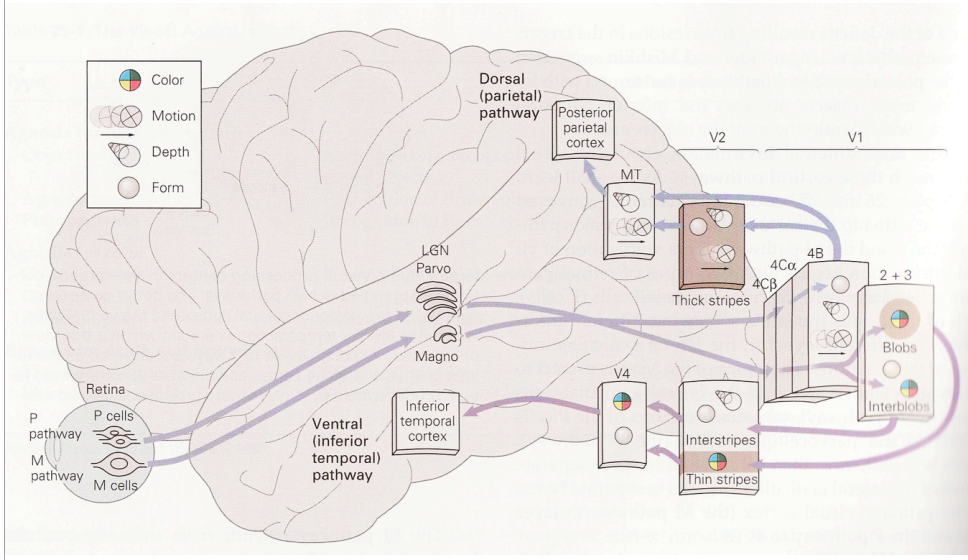
- Non-convex optimization
  ➠local minima: solution depends on where you start...



But convexity may be too restrictive.

Convex problems are mathematically nice and easier, but real-world hard problems may require non-convex models.

Example of a deep architecture made of multiple layers, solving complex problems...



The promises of learning deep architectures

- Representational power of functional composition.
- Shallow architectures
  (NNets with one hidden layer, SVMs, boosting, ...)
  can be universal approximators...
- But may require exponentially more nodes than corresponding deep architectures (see Bengio 2007).
- ➠ statistically more efficient to learn small deep architectures (fewer parameters) than fat shallow architectures.

The notion of Level of Representation