PDEEC – Machine Learning 2018/19

Lecture - Introduction to Neural Networks

Jaime S. Cardoso
jaime.cardoso@inesctec.pt

INESC TEC and Faculdade Engenharia, Universidade do Porto

Nov. 22, 2018

What are Neural Networks?

1. Neural Networks (NNs) are networks of neurons, for example, as found in real (i.e. biological) brain
2. Artificial Neurons are crude approximations of the neurons found in brains. They may be physical devices, or purely mathematical constructs
3. Artificial Neural Networks (ANNs) are networks of Artificial Neurons, and hence constitute crude approximations to parts of real brains. They may be physical devices, or simulated on conventional computers
4. From a practical point of view, an ANN is just a parallel computational system consisting of many simple processing elements connected together in a specific way in order to perform a particular task.

Why are Artificial Neural Networks worth studying?

1. They are extremely powerful computational devices (Turing equivalent, universal computers).
2. Massive parallelism makes them very efficient.
3. They can learn and generalize from training data - so there is no need for enormous feats of programming.
4. They are particularly fault tolerant - this is equivalent to the "graceful degradation" found in biological systems.
5. They are very noise tolerant - so they can cope with situations where normal symbolic systems would have difficulty.

What are Artificial Neural Networks used for?

As with the field of AI in general, there are two basic goals for neural network research:

- **Brain modelling**: The scientific goal of building models of how real brains work. This can potentially help us understand the nature of human intelligence, formulate better teaching strategies, or better remedial actions for brain damaged patients.
- **Artificial System Building**: The engineering goal of building efficient systems for real world applications. This may make machines more powerful, relieve humans of tedious tasks, and may even improve upon human performance.

These should not be thought of as competing goals. We often use exactly the same networks and techniques for both. Frequently progress is made when the two approaches are allowed to feed into each other. There are fundamental differences though, e.g. the need for biological plausibility in brain modelling, and the need for computational efficiency in artificial system building.

Learning in Neural Networks

What is a neural network able to do?

One of the most powerful features of neural networks is their ability to learn and generalize from a set of training data. They adapt the strengths/weights of the connections between neurons so that the final output activations are correct.

1. Regression / approximation (examples: data/system/process modelling, prediction, control, image compression, etc
2. classification (examples: image/handwritten/speech recognition, robotic vision, ECG/EEG diagnosis)
3. Reinforcement learning (i.e. learning with limited feedback)
4. clustering
5. optimization: can be formulated as a neuraldynamic process whose behaviour is determined by learning rules and whose stable states provide solutions to the problem

Projection Pursuit Regression

model: $f(\mathbf{x}) = \sum_{m=1}^{M} g_m(\mathbf{w}_m^T \mathbf{x})$

- additive model on the derived features $\mathbf{w}_m^T \mathbf{x}$ rather than in the original inputs
- sum of nonlinear functions of linear combinations of the inputs
- the functions g_m are unspecified and are estimated along with the directions $\mathbf{w}_m$ using some appropriate method:
  the basis functions are free to adapt to the data
- universal approximator: if M is taken arbitrarily large, for appropriate choice of g_m the PPR model can approximate any continuous function

Projection Pursuit Regression

fitting a PPR model

model: $f(\mathbf{x}) = \sum_{i=1}^{M} g_m(\mathbf{w}_m^T \mathbf{x})$

$$\text{argmin}_{\mathbf{w}_m, g_m} \sum_{n=1}^{N} [y_n - \sum_{m=1}^{M} g_m(\mathbf{w}_m^T \mathbf{x}_n)]^2$$

- a 2-step fitting method, iterated until convergence:
  1. given the direction $\mathbf{w}$ we have a smoothing problem. Apply any model, like smoothing splines to estimate g
  2. given g use a quasi-Newton method to update the vector $\mathbf{w}$
- the number of terms M is usually estimated as part of this fitting method. (Another option is to use cross-validation or equivalent)
- PPR uses nonparametric functions g_m
  - a PPR model typically uses just a few terms ($M = 5$ or 10, for example)

Neural Networks

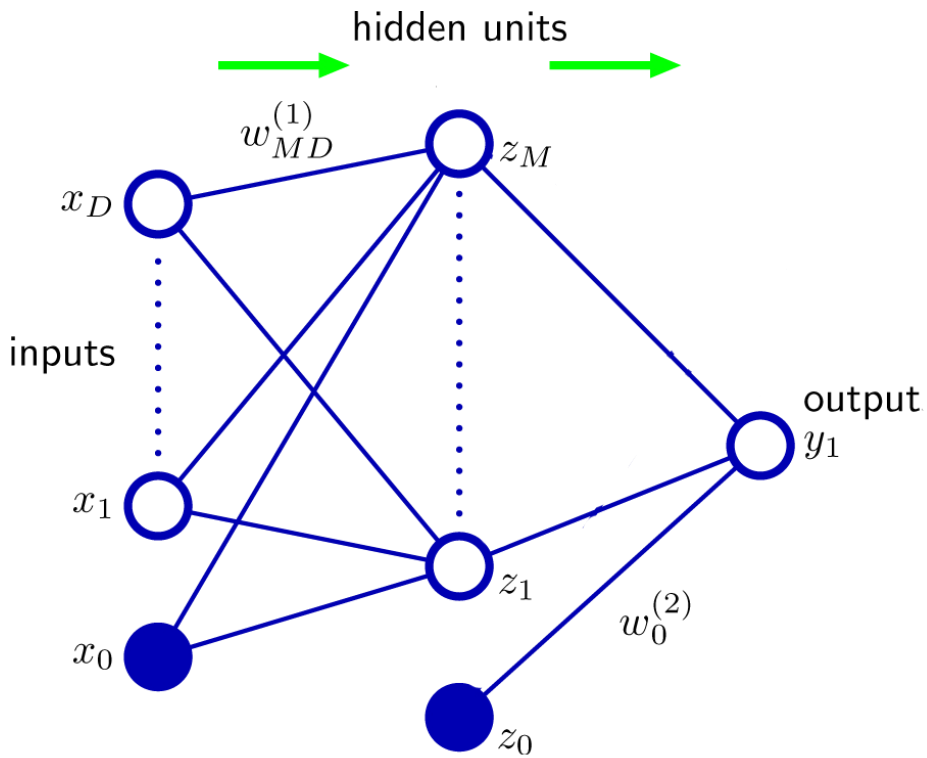
- we have studied models comprising linear combinations of **fixed** basis functions
- like in PPR, the basis functions in neural networks are unspecified and are estimated along with the directions $\mathbf{w}_m$ using some appropriate method:
  the basis functions are free to adapt to the data
- NNs use a fix number of basis functions in advance but allow them to be adaptive.
- basis functions in NN are far simpler that in PPR, parametrized with just a few parameters

Feed-forward Neural Networks

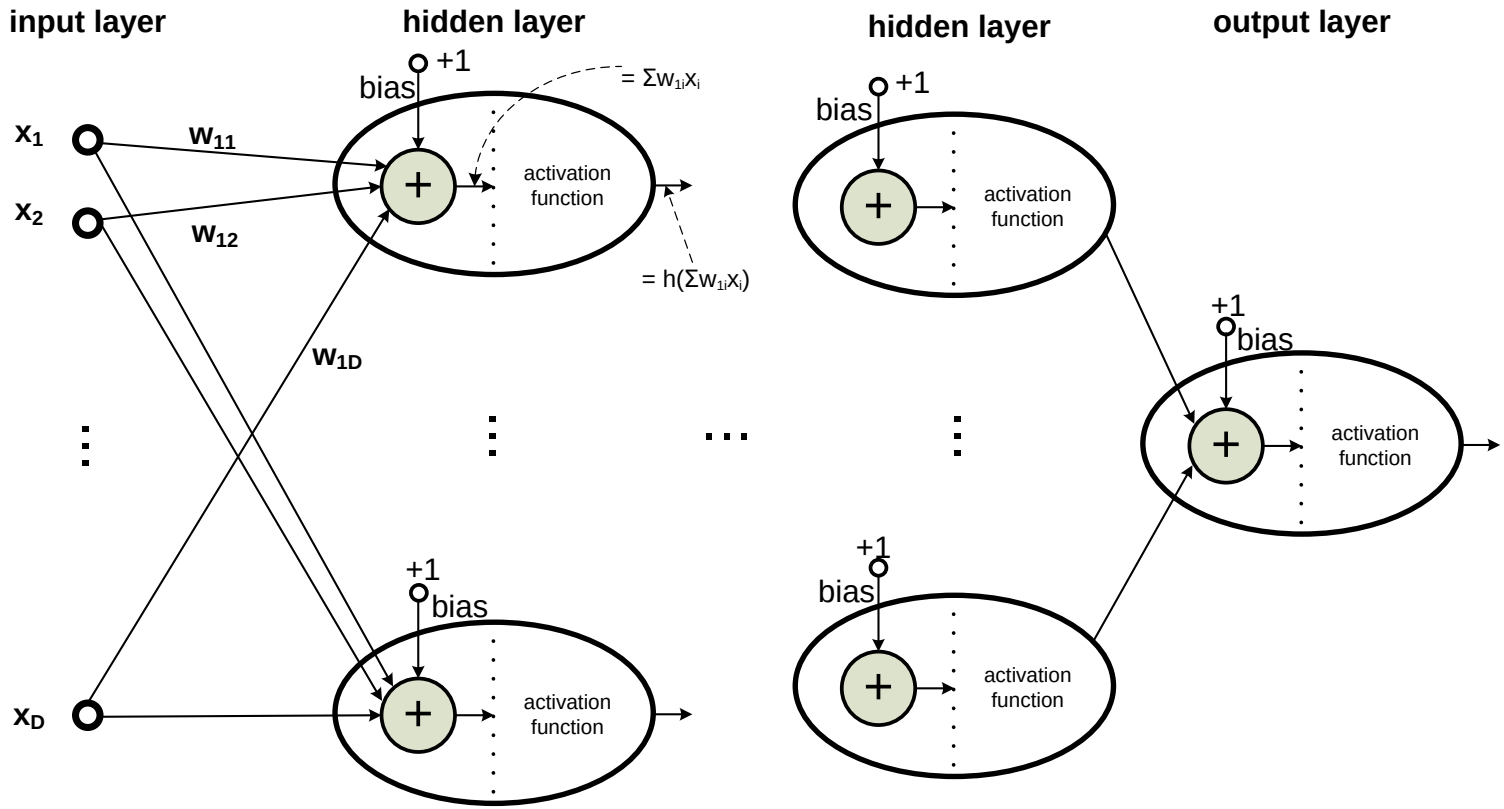
Multi-layer perceptron Neural Networks

- Extension of the model 'linear combination of fixed nonlinear basis functions'

$$y(\mathbf{x}, \mathbf{w}) = f(\sum_{j=1}^{M} w_j h_j(\mathbf{x}))$$



Multi-layer perceptron Neural Networks



Feed-forward Neural Networks

Multi-layer perceptron Neural Networks

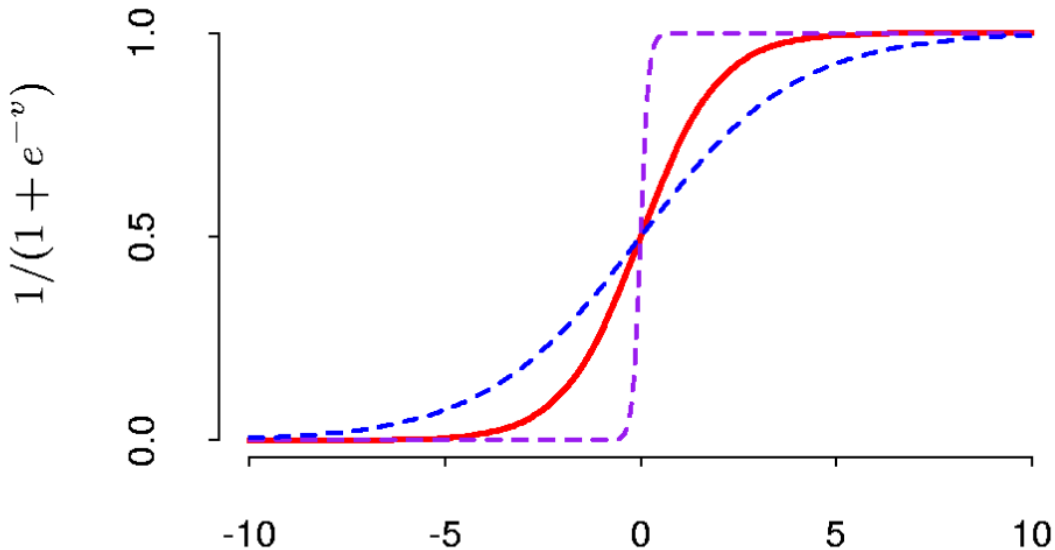
- The activation function is usually chosen to be the sigmoid $\sigma(v) = \frac{1}{1+e^{-v}}$
- notice that if σ is the identity function then the entire model collapses to a linear model in the inputs
- the NN model with one hidden layer has exactly the same form as the PPR model
  - the lower complexity of the basis functions in NNs allows using more of such functions

Feed-forward Neural Networks

Multi-layer Neural Networks

Common activation function

- $step - function(v) = \begin{cases} 1 & v \geq 0 \\ 0 & otherwise \end{cases}$
- logistic sigmoid function $\sigma = \frac{1}{1+e^{-v}}$



- $tanh(v) = \frac{e^{2v}-1}{e^{2v}+1} = 2\sigma(2v) - 1$
- linear on the output for regression

Feed-forward Neural Networks

Multi-layer Neural Networks

Universal approximation theorem:
Given any $\xi > 0$ and any continuous function $f : [0, 1]^d \to \mathbf{R}$, there exists a single-hidden-layer perceptron, with bounded and monotone-increasing continuous neuron activation function, which is able to approximate f within ξ mean square error accuracy.

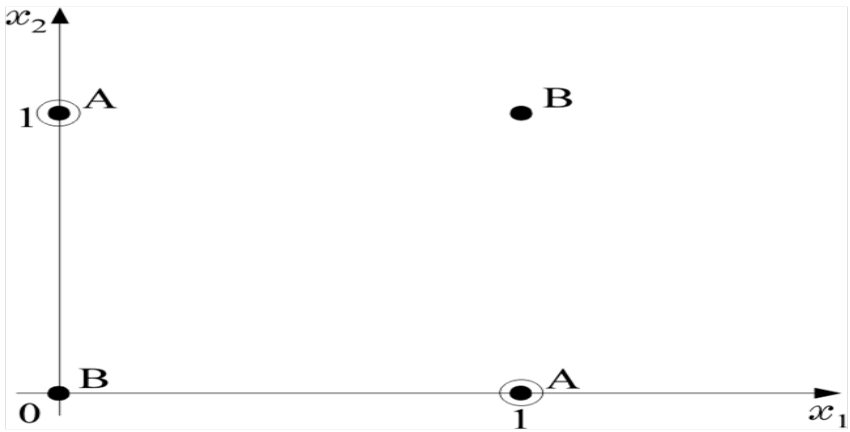
- the problem is how to find suitable parameter values given a training data

Feed-forward Neural Networks

Multi-layer perceptron Neural Networks

The XOR problem:

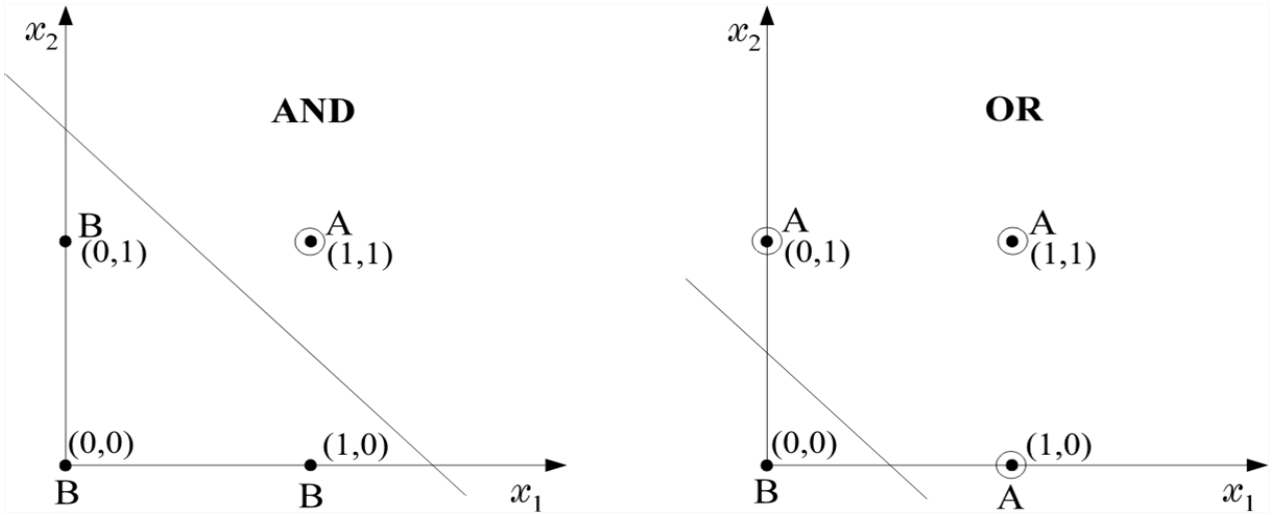
x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B



Feed-forward Neural Networks

Multi-layer perceptron Neural Networks

- There is no single line (hyperplane) that separates class A from class B. On the contrary, AND and OR operations are linearly separable problems

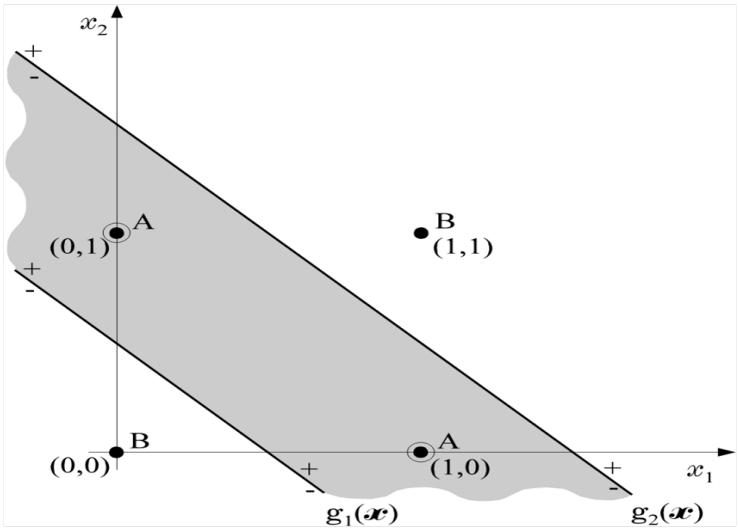


Feed-forward Neural Networks

Multi-layer perceptron Neural Networks

The Two-Layer Perceptron

- For the XOR problem, draw two, instead, of one lines



- The computations of the first phase perform a mapping $\mathbf{x} \rightarrow \mathbf{y} = [y_1 y_2]^T$

1st phase				2nd phase
x_1	x_2	y_1	y_2	
0	0	0(-)	0(-)	B(0)
0	1	1(+)	0(-)	A(1)
1	0	1(+)	0(-)	A(1)
1	1	1(+)	1(+)	B(0)

Feed-forward Neural Networks

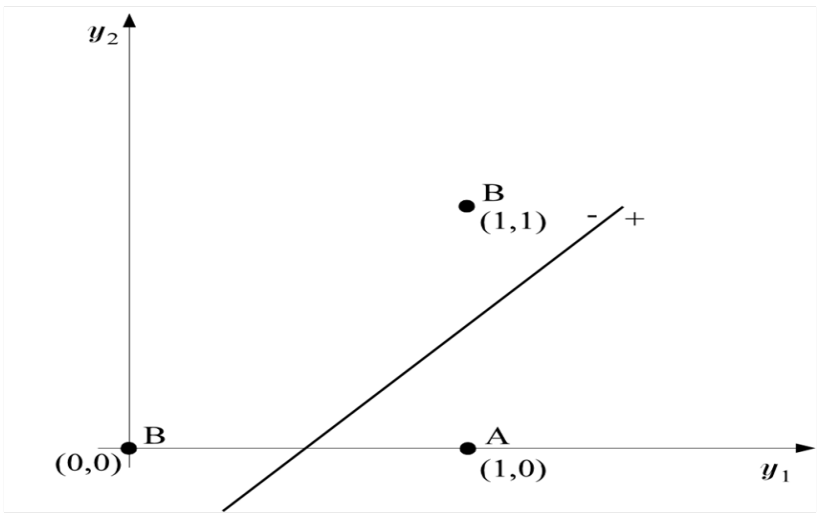
Multi-layer perceptron Neural Networks

The Two-Layer Perceptron

- The computations of the first phase perform a mapping $\mathbf{x} \rightarrow \mathbf{y} = [y_1 \ y_2]^T$

1st phase				2nd phase
x_1	x_2	y_1	y_2	
0	0	0(-)	0(-)	B(0)
0	1	1(+)	0(-)	A(1)
1	0	1(+)	0(-)	A(1)
1	1	1(+)	1(+)	B(0)

- The decision is now performed on the transformed data. This can be performed via a second line, which can also be realized by a perceptron.

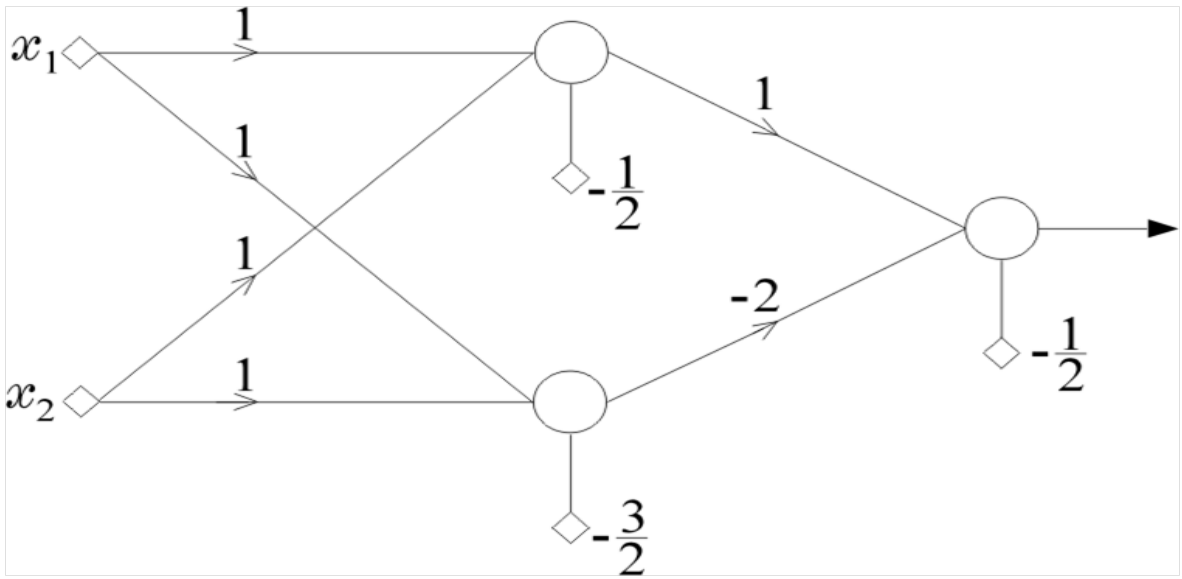


Feed-forward Neural Networks

Multi-layer perceptron Neural Networks

The Two-Layer Perceptron

- Computations of the first phase perform a mapping that transforms the nonlinearly separable problem to a linearly separable one.
- architecture:

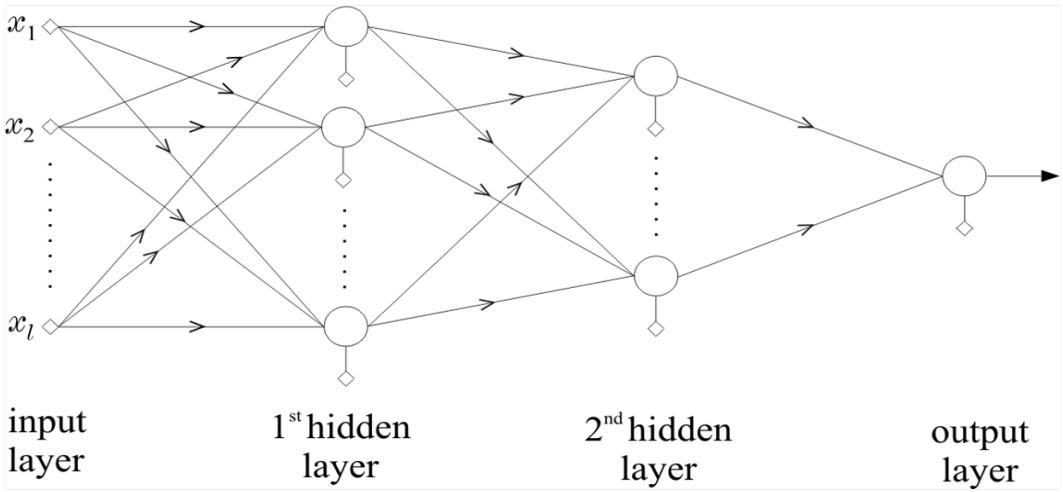


Feed-forward Neural Networks

Multi-layer perceptron Neural Networks

The Three-Layer Perceptron

- The output neuron of a two-layer perceptron realizes a hyperplane in the transformed y space, that separates some of the vertices from the others. Thus, the two layer perceptron has the capability to classify vectors into classes that consist of unions of polyhedral regions. But NOT ANY union. It depends on the relative position of the corresponding vertices.
- The Three-Layer Perceptron architecture:



- This is capable to classify vectors into classes consisting of ANY union of polyhedral regions.

Feed-forward Neural Networks

Multi-layer perceptron Neural Networks

- we have focused on the potential capabilities of a three-layer perceptron
- we have to resort to learning algorithms to learn the weights from the available training data
- the discontinuity of the step (activation) function, prohibiting differentiation with respect to the unknown parameters ($\mathbf{w}$) causes difficulties to the optimization procedure
- we move away from the step function and use other activation functions

Multi-layer perceptron Neural Networks

Network Training

- we have viewed neural networks as a general class of parametric nonlinear functions
- as with other models, and given a training data, we determine the network parameters by minimizing some error function
- in regression we define the error function as $E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} ||y(\mathbf{x}_n, \mathbf{w}) - y_n||^2$
- how to optimize the parameters $\mathbf{w}$?
- impossible to find analytically the points where $\nabla E(\mathbf{w}) = 0$

Multi-layer perceptron Neural Networks

M(C)LE Training for Neural Networks

- Consider regression problem $f : \mathbf{x} \rightarrow y$, for scalar y

$y = f(x) + \varepsilon$ ← assume noise $N(0,\sigma_\varepsilon)$, iid

deterministic

- Let's maximize the conditional data likelihood

$$W \leftarrow \arg\max_W \ln \prod_l P(Y^l | X^l, W)$$

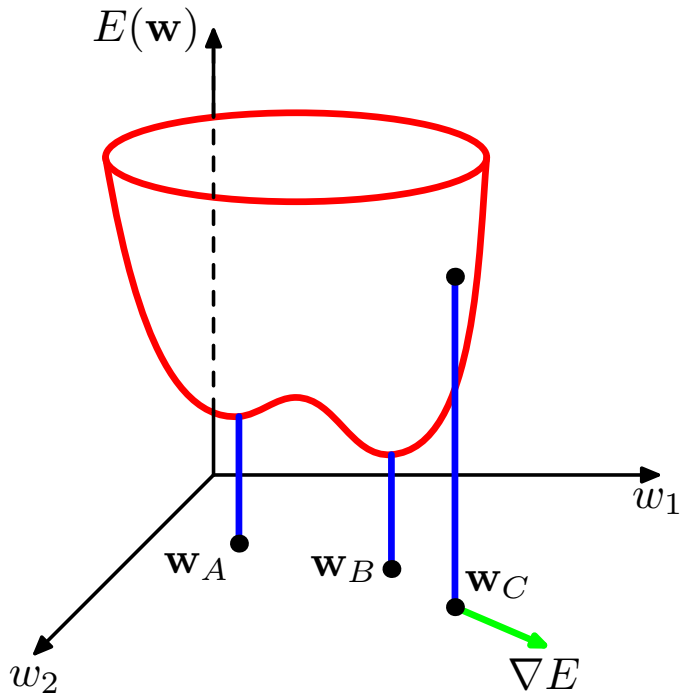
$$W \leftarrow \arg\min_W \sum_l (y^l - \hat{f}(x^l))^2$$

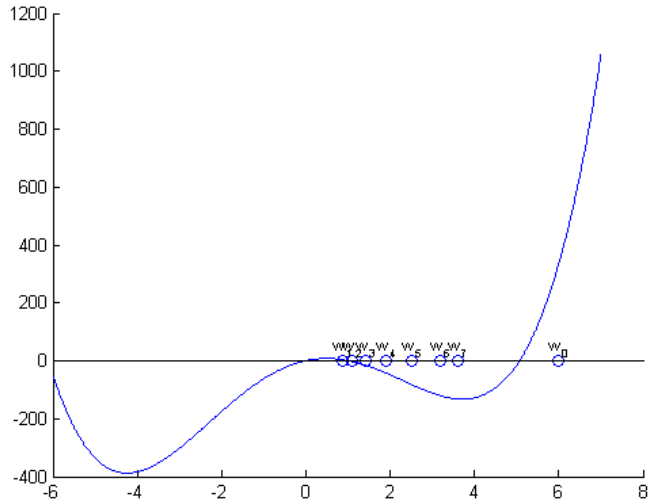
Learned neural network

Multi-layer perceptron Neural Networks

Parameter optimization

Gradient descent scheme





- most techniques involve chosing some initial guess $\mathbf{w}^{(0)}$ and improve the approximation in each iteration, moving on the weight space: $\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}$
- in many algorithms $\Delta\mathbf{w}^{(\tau)}$ is based on the gradient information: $\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E(\mathbf{w}^{(\tau)})$
- $E(\mathbf{w} + \delta\mathbf{w}) \approx E(\mathbf{w}) + \delta\mathbf{w}^T\nabla E(\mathbf{w})$

Multi-layer perceptron Neural Networks

Parameter optimization

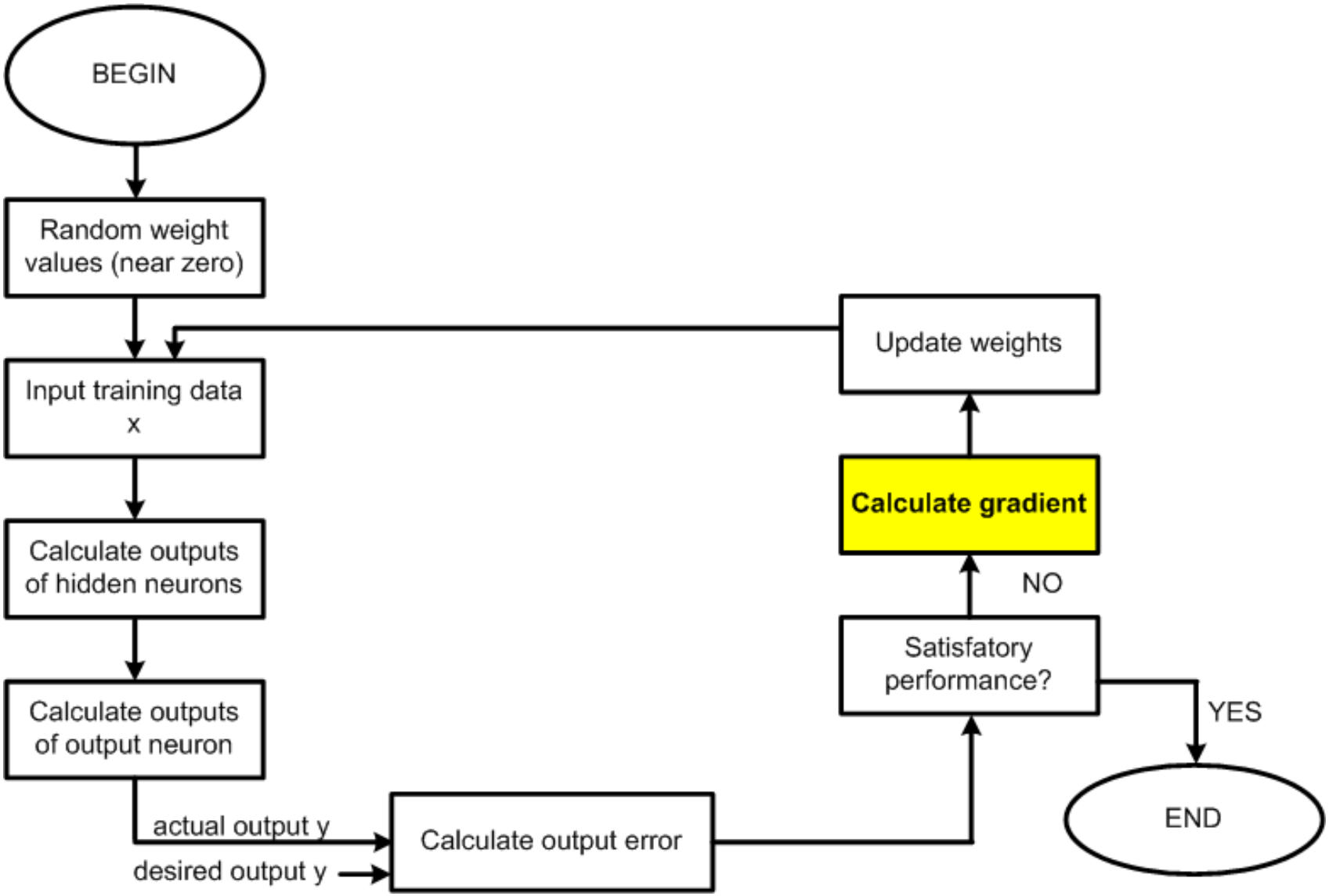
Gradient descent scheme

- batch methods: use the whole data set to evaluate ∇E
- sequencial gradient descent or stochastic gradient descent: makes an update to the weight vector based on one data point at a time:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)})$$

Multi-layer perceptron Neural Networks

Network training



Error Backpropagation

Computationally efficient method for evaluating the derivatives of the error function.

- see section 5.3.1 and 5.3.2 of Bishop
  1. apply the input vector to the network and forward propagate through the network to find the activations of all hidden and output units
  2. evaluate δ for all output units
  3. backpropagate δ to obtain δ for each hidden unit in the network
  4. evaluate the derivatives
- $h(a) = tanh(a) \Rightarrow h'(a) = 1 - h(a)^2$
- $h(a) = \sigma(a) \Rightarrow h'(a) = \sigma(a)(1 - \sigma(a))$

Gradient descent scheme with Error Backpropagation

1. Initialize all weights to small random numbers
2. Until convergence, Do
   - 2.1 Input the training example to the network and compute the neurons outputs and network output(s)
   - 2.2 For each output unit k $\delta_k \leftarrow \frac{\partial E}{\partial a}$
     a: input to the output activation function
     h: output activation function
     $\delta_k \leftarrow \frac{\partial E}{\partial a} = (z - y)z(1 - z)$
     $z = h(a)$, and assuming logistic sigmoid function as the activation function and the square of the error.
   - 2.3 for each hidden unit $\delta_k \leftarrow h'(a) \sum w_{k,j}\delta_j$
   - 2.4 $\frac{\partial E}{\partial w_{ij}} = z_i \delta_j$
     z_i: input to edge w_{ij}
     δ_j: error at the output of edge w_{ij}
   - 2.5 update the $\mathbf{w}$ vector

Multi-layer perceptron Neural Networks

The cost function choice

Binary classification

- output activation function: logistic sigmoid function $\sigma = \frac{1}{1+e^{-v}}$
- interpreted as the probability $p_{x,w}$ of belonging to class $\mathcal{C}_1$
- the probability of observing y_n is $p_{x,w}^{y_n}(1-p_{x,w})^{1-y_n}$ (Bernoulli distribution)
- the probability of observing the training data is $\prod_{n=1}^{N} p_{x_n,w}^{y_n}(1-p_{x,w})^{1-y_n}$
- the log likelihood function is given by

$$\sum_{n=1}^{N} y_n \log p_{x_n,w} + (1-y_n)\log(1-p_{x_n,w})$$

- the error function is the cross-entropy function:

$$E(\mathbf{w}) = -\sum_{n=1}^{N} y_n \log p_{x_n,w} + (1-y_n)\log(1-p_{x_n,w})$$

Multi-layer perceptron Neural Networks

The cost function choice

multiclass classification (nominal data):

- output coded as 1-of-K.
- Network with K outputs. Output k interpreted as the probability of class $\mathcal{C}_k$. Therefore, we should have $\sum_{k=1}^{K} p_k = 1$.
- the probability of observing $y_n = (0, \ldots, 0, 1, 0, \ldots, 0)$ is $\prod_{k=1}^{K} p_k^{y_n(k)}$
- the probability of observing the training data is $\prod_{n=1}^{N} \prod_{k=1}^{K} p_k^{y_n(k)}$
- the log likelihood function is given by

$$\sum_{n=1}^{N} \sum_{k=1}^{K} y_n(k) \log p_{k,x_n,w} + (1 - y_n(k)) \log(1 - p_{k,x_n,w})$$

- the error function is:

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} y_n(k) \log p_{k,x_n,w} + (1 - y_n(k)) \log(1 - p_{k,x_n,w})$$

- the output activation function is the softmax function: $p_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}$

starting values

- if the weights are near zero, the operative part of the sigmoid is roughly linear, and hence the neural network collapses into an approximately linear model.
- use of exact zero values leads to zero derivatives and the algorithm never moves
- starting values for the weights are usually chosen to be random values near zero. Hence the model starts out nearly linear, and becomes nonlinear as the weights increase.
- nonlinearities are introduced during training where needed.
- starting with large weight values often leads to poor solutions

Scaling the inputs

- it is usual to standardize all inputs to have mean zero and standard deviation one.
  - or between 0 and 1... or between -1 and 1
- ensures that all inputs are treated equally in the regularization process
- facilitates the choice of a meaningful range for the random starting weights. It is typical to take random uniform weights over $[-0.7, +0.7]$, for standardized inputs

Multi-layer perceptron Neural Networks

Some issues in training NNs

Size of the network

From [4]:

- choose the size of the network large enough and leave the training to decide about the weights is a naive approach.
- the number of free parameters should be a) large enough to learn what makes similar the example within the same class and what makes one class different from the other; b) small enough so as not to be able to learn the underlying differences among the data of the same class (overfit)

From [3]:

- generally speaking it is better to have too many hidden units than too few: with too few the model might not have enough flexibility to capture the nonlinearities in the data; with too many hidden units, the extra weights can be shrunk toward zero if appropriate regularization is used.
- use cross-validation to choose the regularization parameter
- use cross-validation to choose the number of hidden units? Maybe unnecessary.
- choice of the number of hidden layers is guided by background knowledge and experimentation.

Multi-layer perceptron Neural Networks

MAP Training for Neural Networks

- Consider regression problem $f : \mathbf{x} \rightarrow y$, for scalar y

$y = f(x) + \varepsilon$ ← noise $N(0,\sigma_\varepsilon)$

deterministic

Gaussian $P(W) = N(0,\sigma I)$

$$W \leftarrow \arg\max_W \ln P(W) \prod_l P(Y^l|X^l, W)$$

$$W \leftarrow \arg\min_W \left[c \sum_i w_i^2\right] + \left[\sum_l (y^l - \hat{f}(x^l))^2\right]$$

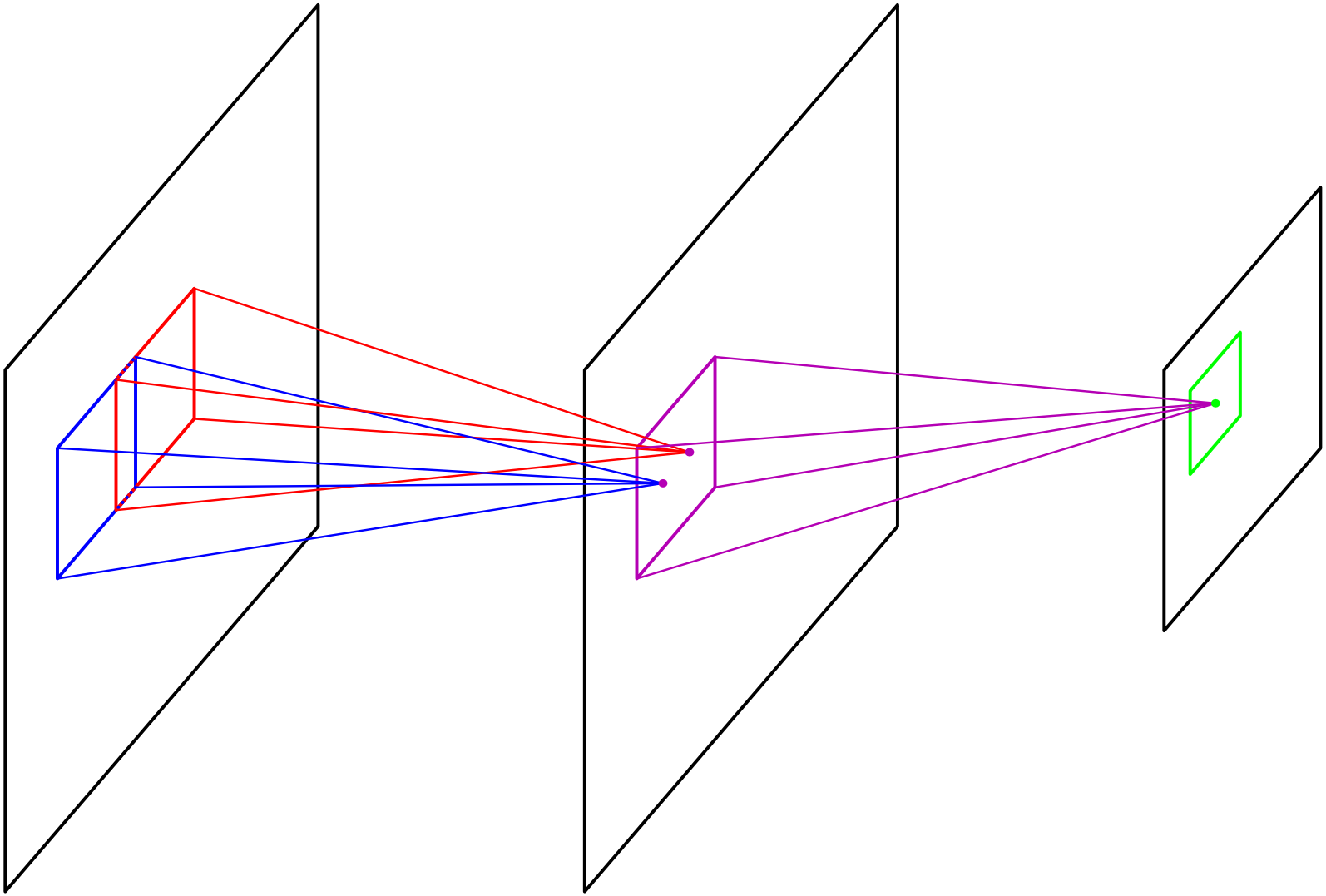
$\ln P(W) \leftrightarrow c \sum_i w_i^2$

Regularization and the problem of overfitting

- weight decay: $\hat{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$
  for consistency over scaling transformations the regularization term should be $\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \sum_{w\in\mathcal{W}_2} w^2$, with biases excluded from the summations.
- early stopping
- invariances:
  - extract features that are invariant under the required transformation
  - augment the training set using replicas of the training patterns, transformed according to the desired imvariances
  - add a regularization term to the error function penalizing changes in the model output when the input is transformed
  - build invariances properties into the structure of the neural network (shared weights, etc)

Multi-layer perceptron Neural Networks

incorporating prior knowledge in the architecture



Input image

Convolutional layer

Sub-sampling layer

Multi-layer perceptron Neural Networks

Bayesian Neural Networks

Other Neural Networks models

other Neural Networks models

feed forward NNs: radial basis functions (RBF):

- RBF networks have structure similar to MLP, similar applications but different neuron model, different learning algorithm
- $g(x) = w_0 + \sum_{i=1}^{M} w_i \exp\left(-\gamma_i (\mathbf{x} - c_i)^T (\mathbf{x} - c_i)\right)$
- the activation responses of the nodes are of a local nature
- there are two groups of free parameters: $\{w_i\}$ and $\{c_i, \gamma_i\}$
- there are different learning strategies

Other Neural Networks models

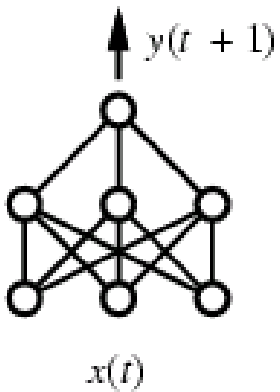
other Neural Networks models

- feed forward NNs: RBF
  RBF networks have structure similar to MLP, similar applications but different neuron model, different learning algorithm
- recurrent networks: feedback look between output neurons and input nodes.
  Suitable for processing temporal data due to their internal representation of time
  - MLP can also be used for time series prediction
  - extension of auto-regressive time series modelling
  - window size? sample rate?
- self organizing maps (SOMs)

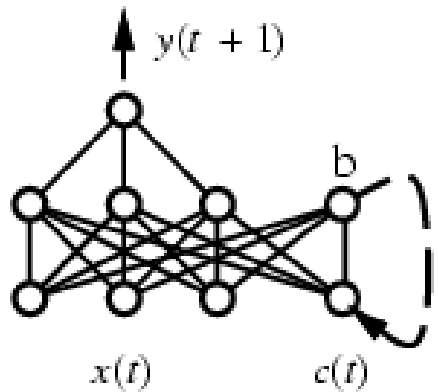
Other Neural Networks models

Training Networks on Time Series

- Suppose we want to predict next state of world
  - and it depends on history of unknown length
  - e.g., robot with forward-facing sensors trying to predict next sensor reading as it moves and turns
- Idea: use hidden layer in network to capture state history



(a) Feedforward network

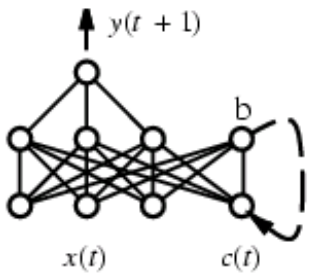


(b) Recurrent network

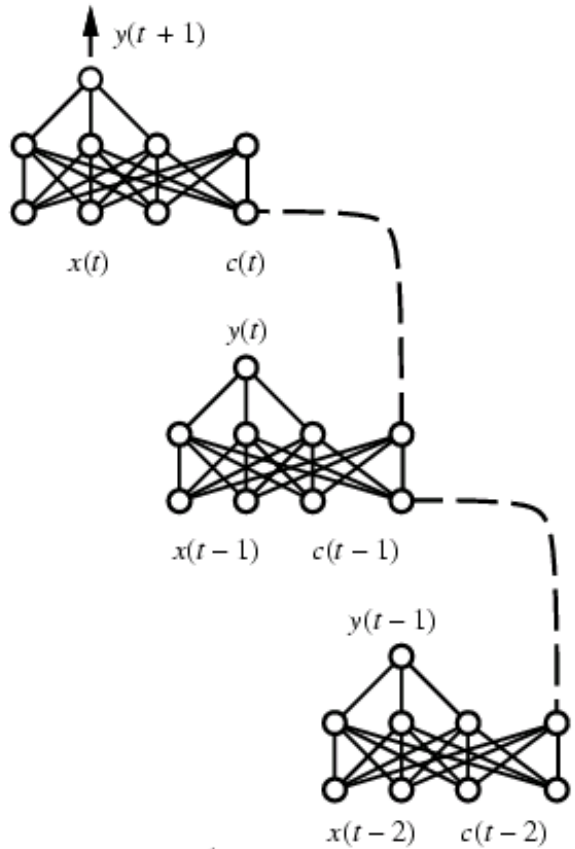
Other Neural Networks models

Training Networks on Time Series

How can we train recurrent net??



(b) Recurrent network



(c) Recurrent network unfolded in time

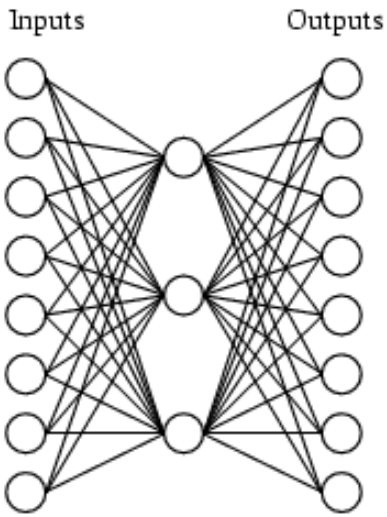
Other Neural Networks models

Training Networks on Time Series

- Training recurrent networks by unfolding in time is not very reliable
- Newer idea
  - randomly create a fixed (untrained) recurrent network
  - then train a classifier whose input is the network internal state, output is whatever you wish
  - suggested [Maas, et al., 2001] as model of cortical microcolumns
  - "liquid state" or "echo state" analogy to water reservoir
  - NIPS 2007 Workshop on Liquid State Machines and Echo State Networks

Other Neural Networks models

Learning Hidden Layers Representations



A target function:

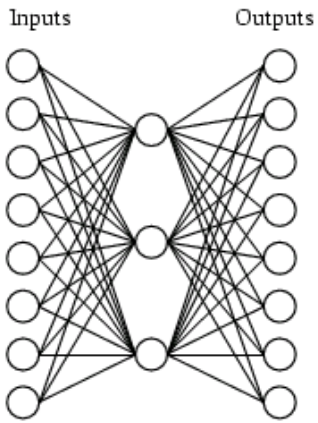
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned??

Other Neural Networks models

Learning Hidden Layers Representations

A network:



Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Artificial Neural Networks

Summary

- Highly non-linear regression/classification
- Vector-valued inputs and outputs
- Potentially millions of parameters to estimate
- Hidden layers learn intermediate representations
- Actively used to model distributed computation in brain
- Gradient descent, local minima problems
- Overfitting and how to deal with it
- Many extensions

References


John A. Bullinaria
Introduction to Neural Networks - Course Material and Useful Links,
http://www.cs.bham.ac.uk/ jxb/inn.html


Bishop
Pattern recognition and machine learning,
Book.


Trevor Hastie and Robert Tibshirani and Jerome Friedman
The elements of statistical learning,
Springer.


Sergios Theodoridis and Konstantinos Koutroumbas
Pattern recognition
Elsevier (book)