Deep Learning

Course organisation

- **Arben Cela** (a.cela@esiee.fr, office 5453)
- Courses 4h:
  - Introduction
  - Perceptron
  - Adaline
  - Multi-Layer Neural network: Static and dynamic Backprapagation
  - Applications on temporal series learning
  - Annexe: Least mean Square and Levenberg-Marquardt algorithm
- Tutorials work 12h:
  - Adaptive Filtering of Noisy Signal using Adaline Neural Network (4h)
  - Data analitics and prediction (4h)
  - Chickenpox Propagation Prediction or Inverted Pendulum Control (4h)
- Grading: Tutorials reports
- References
  1. M. T. Hagan, H. B. Demuth, M.H. Beale and O. De Jesus, *Neural Network Design*, Second edition, PWS Publishing, Boston
  2. F. Chollet, *Deep Learning with Python*, Manning Publication, 2018
  3. M. T. Hagan and H. B. Demuth and O. De Jesus, *An introduction to the use of neural networks in control systems*, International Journal of Robust and Nonlinear Control, John Wiley & Sons, 2002, Vol. 12, pp. 959-985

Deep Learning

Application domains

- Aerospace: High performance aircraft autopilots, flight path simulations, aircraft control systems, autopilot enhancements, aircraft component simulations, aircraft component fault detectors
- Automotive: Automobile automatic guidance systems, fuel injector control, automatic braking systems, misfire detection, virtual emission sensors, warranty activity analyzers
- Banking: Check and other document readers, credit application evaluators, cash forecasting, firm classification, exchange rate forecasting, predicting loan recovery rates, measuring credit risk
- Medical: Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, emergency room test advisement
- Telecommunication: Image and data compression, automated information services, real-time translation of spoken language, customer payment processing system
- . . .:

Deep Learning

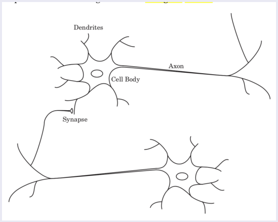
Don’t believe the short-term hype

- The pionneer of AI Marvin Minsky claimed in 1967: *Within a generation . . . the problem of creating Artificial Intelligence will substantially be solved...!*
- Three years later, in 1970, he claimed: *In from three to eight years we will have a machine with the general intelligence of an average human being...!*
- A few years later start of the first AI winter and the second one was around 1985 on the *Expert Systems*.
- Neural Network beginning at the end of 1980...

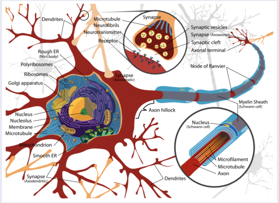
Deep Learning

Simple Biological Neuron

- Brain has 10^{11} neurons which has 10^4 connections.
- Dendrites are tree-like receptive networks of nerve fibers that carry electrical signals into the cell body.
- Cell body sums and thresholds incoming signals.
- Axon is a single long fiber that carries the signal from the cell body out to other neurons.



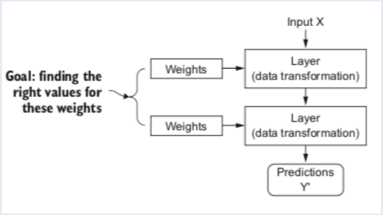
- Synapse is the point of contact between an axon of one cell and a dendrite of another cell

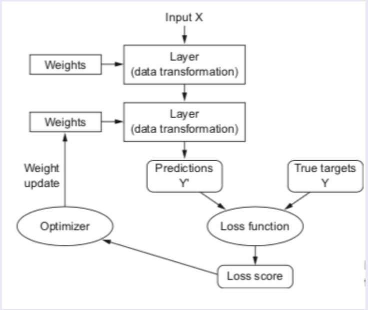


Deep Learning

Anatomy of a Neural Network

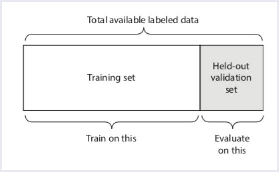
- *Layers*, which are combined into a *network or model*
- The *input data* and corresponding *targets*
- The *loss function*, which defines the feedback signal used for learning
- The *optimizer*, which determines how learning proceeds

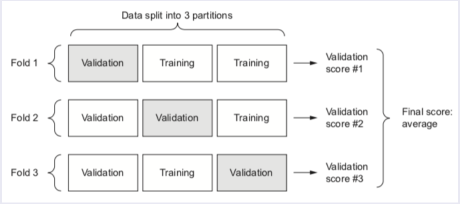

Input X
Goal: finding the right values for these weights
Weights
Layer (data transformation)
Weights
Layer (data transformation)
Predictions Y'



Deep Learning

Data Partitioning: Simple and K-fold Cross-Validation Partitioning





- Data representativeness:⇒ You want both your training set and test set to be representative of the data at hand
- The arrow of time:⇒ If you're trying to predict the future given the past you should preserve the data temporal order (not randomly shuffle your data before splitting), in order to avoid temporal leak: *your model will effectively be trained on data from the future*.
- Redundancy in your data:⇒ Shuffling a redundant data set before splitting it into a training set and a validation set will result in redundancy between the training and validation sets. In effect, you'll be testing on part of your training data, which is the worst thing you can do! Make sure your training set and validation set are disjoint.

Deep Learning

Data preprocessing for neural networks

- Vectorization
- Normalization
- Handling missing values
- Feature extraction

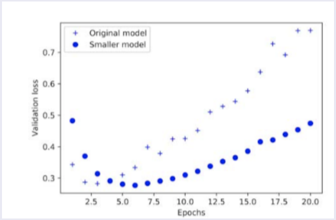
Learning Workflow

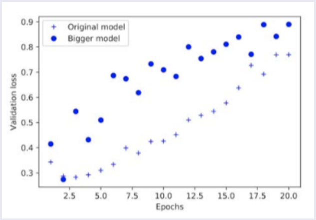
- Define your training data: input tensors and target tensors
- Define a network of layers (or model) that maps your inputs to your targets
- Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor
- Iterate on your training data by calling the learning (fit()-in Keras) method of your model

Deep Learning

Reduce Overfitting

- The fundamental issue in deep learning is the tension between optimization and generalization: The goal of the game is to get good generalization
- Model underfit:$\Rightarrow$ Optimization and generalization are correlated and lower loss in trained date imply lower loss in test data
- Model overfit:$\Rightarrow$ Generalization stops improving, and validation metrics stall and then begin to degrade

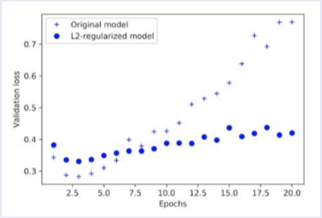


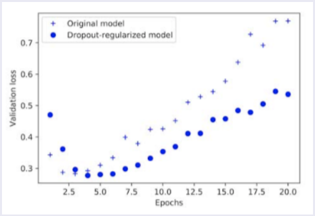


Deep Learning

Reduce Overfitting

- Prevent learning misleading:⇒ Best solution is to get more training data
- NN size regularization:⇒ Reduce the size of NN in order to focus learning on the most prominent patterns, which have a better chance of generalizing well
- Weight regularization:⇒ Adding to the loss function of the network a cost associated with having large weights (L1 or L2 norm)
- Adding dropout:⇒ Applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.





Deep Learning

Classification and regression glossary

- Sample or input:⇒One data point that goes into your model
- Prediction or output:⇒What comes out of your model
- Target:⇒The truth. What your model should ideally have predicted
- Prediction error or loss value:⇒A measure of the distance between your model's prediction and the target
- Classes:⇒A set of possible labels to choose from in a classification problem (...cat and dog pictures, "dog" and "cat" are the two classes)
- Label:⇒A specific instance of a class annotation in a classification problem.
- Ground-truth or annotations:⇒All targets for a dataset, typically collected by humans.
- Binary classification:⇒A classification task where each input sample should be categorized into two exclusive categories
- Multiclass classification:⇒A classification task where each input sample should be categorized into more than two categories (classifying handwritten digits).

Deep Learning

Classification and regression glossary

- Multilabel classification:⇒A classification task where each input sample can be assigned multiple labels([a])
- Scalar regression:⇒A task where the target is a continuous scalar value
- Vector regression:⇒A task where the target is a set of continuous values (ex: a continuous vector)([b])
- Mini-batch or batch:⇒A small set of samples (typically between 8 and 128) that are processed simultaneously by the model

[a]for instance, a given image may contain both a cat and a dog and should be annotated both with the ”ca” label and the”do” label. The number of labels per image is usually variable

[b]If you're doing regression against multiple values (such as the coordinates of a bounding box in an image), then you?re doing vector regression

Deep Learning

Simple Perceptron Architecture

- p is the input signal
- w, b are the input and biased weights
- n is the summer output or net input
- f is the neuron transfer function
- a is the neuron output or the activation signal

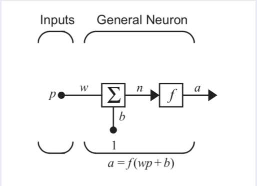
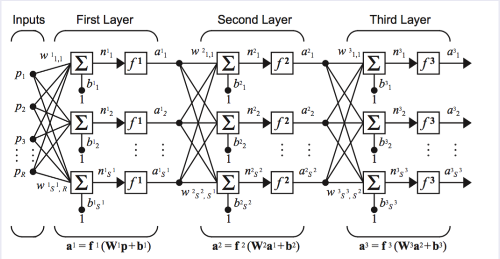


Figure 2.1 Single-Input Neuron

Multi-Layer Perceptron Architecture



Deep Learning

Perceptron Architecture

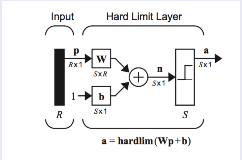


Figure 4.1 Perceptron Network

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}, {}_i w = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}, W = \begin{bmatrix} {}_1 w^T \\ {}_2 w^T \\ \vdots \\ {}_S w^T \end{bmatrix}$$

- This allows us to write the i^{th} element of the network output vector as:$a_i = hardlim(n_i) = hardlim({}_i w^T p + b_i)$ where $hardlim(n) = 1$ if $n \geq 0$ and zero for $n < 0$.
- Therefore, if the inner product of the i^{th} row of the weight matrix with the input vector is greater than or equal to $-b_i$, the output will be 1, otherwise the output will be 0. Thus each neuron in the network divides the input space into two regions.

Deep Learning

Example: Two input/single output Perceptron Architecture

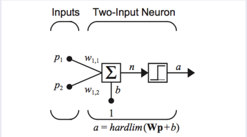


Figure 4.2 Two-Input/Single-Output Perceptron

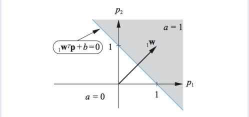
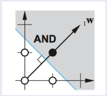


Figure 4.3 Decision Boundary for Two-Input Perceptron

- $a = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b)$
- The decision boundary is determined by the input vectors for which the net input n is zero: $n =_1 w^T p + b = w_{1,1}p_1 + w_{1,2}p_2 + b = 0$
- Assign the following values for the weights and bias: $w_{1,1} = 1, w_{1,2} = 1, b = -1 \Rightarrow n = p_1 + p_2 - 1 = 0$
- The line in (op_1p_2) plan given by $n = p_1 + p_2 - 1 = 0$ separates it in two sub plans and the line cut the axis $0p_1$ at point $(1,0)$ and axis $0p_2$ at point $(0,1)$.
- We can also find the decision boundary graphically. The first step is to note that the boundary is always orthogonal to ${}_1w$.
- After we have selected a weight vector with the correct angular orientation, the bias value can be computed by selecting a point on the boundary and satisfying $n =_1 w^T p + b = 0$.

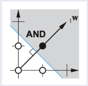
Deep Learning

Example: AND gate two input/single output Perceptron



The input/target pairs for the AND gate are:

$$\left\{p_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\} \left\{p_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 0\right\} \left\{p_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 0\right\} \left\{p_2 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}$$



- We want to have a line that separates the dark circles and the light circles and take as solution the line that falls "halfway" between the two categories of inputs, as shown in the adjacent figure.

- We can also find the decision boundary graphically. The first step is to note that the boundary is always orthogonal to ${}_1w$.
- Next we want to choose a weight vector that is orthogonal to the decision boundary. The weight vector can be any length, so there are infinite possibilities. One choice is: ${}_1w = \begin{bmatrix}2 & 2\end{bmatrix}^T$.
- Finally, we need to find the bias, b . We can do this by picking a point on the decision boundary and satisfying output decision equation. If, for example we chose $p = \begin{bmatrix}1.5 & 0\end{bmatrix}^T$ we find:

$${}_1w^Tp + b = \cancel{{}_1w} = \begin{bmatrix}2 & 2\end{bmatrix}\begin{bmatrix}1.5\\0\end{bmatrix} + b = 0 \Rightarrow b = -3.$$

- We have to test the network on all of the input/target pairs.

Deep Learning

Multilayer Perceptron

- Note that a single-neuron perceptron can classify input vectors into two categories, since its output can be either 0 or 1.
- A multiple-neuron perceptron can classify inputs into many categories. Each category is represented by a different output vector. Since each element of the output vector can be either 0 or 1, there are a total of 2^S possible categories, where S is the number of neurons.

Deep Learning

Function approximation

- It is also instructive to view networks as function approximators. In control systems, for example, the objective is to find an appropriate feedback function $u(t) = f_b(y(t))$ that maps from measured outputs $y(t)$ to control inputs $u(t)$.
- In adaptive filtering the objective is to find a function that maps from delayed values of an input signal to an appropriate output signal

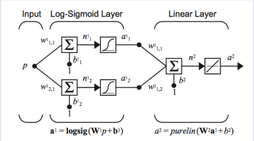


Figure 11.4 Example Function Approximation Network

Application:(Matlab test file nnd11nf.m)

$w^1_{1,1} = 10$, $w^1_{2,1} = 10$, $b^1_1 = -10, b^1_2 = 10$, $w^2_{1,1} = 1$, $w^2_{1,2} = 1$, $b^2 = 0$ we obtain the nominal response given in *Fig*.11.5.

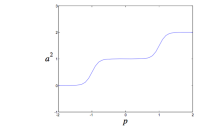


Figure 11.5 Nominal Response of Network of Figure 11.4

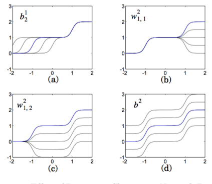


Figure 11.6 Effect of Parameter Changes on Network Response

- Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer. The centers of the steps occur where the net input to a neuron in the first layer is zero: $n^1_1 = w^1_{1,1}p + b^1_1 = 0 \Rightarrow p = -b^1_1/w^1_{1,1} = 10/10 = 1$; $n^1_2 = w^1_{2,1}p + b^1_2 = 0 \Rightarrow p = -b^1_2/w^1_{2,1} = -10/10 = -1$
- Figure 11.6 illustrates the effects of parameter changes on the network response. The blue curve is the nominal response. The other curves correspond to the network response when one parameter at a time is varied over the following ranges: $-1 \leq w^2_{1,1} \leq 1, -1 \leq w^2_{1,2} \leq 1, 0 \leq b^1_2 \leq 20, -1 \leq b^1_2 \leq 1$.

Deep Learning

ADALINE Network

- Bernard Widrow and Marcias Hoff in 1960 proposed ADALINE (**ADA**ptive **LI**near **NE**uron)
- LMS (Least Mean Square) algorithm was proposed as learning rule.
- In the 1980s Bernard Widrow began research on the use of neural networks in adaptive control, using temporal backpropagation, a descendant of his original LMS algorithm.

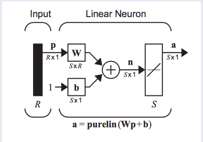


Figure 10.1 ADALINE Network

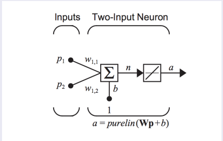


Figure 10.2 Two-Input Linear Neuron

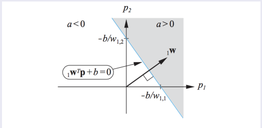


Figure 10.3 Decision Boundary for Two-Input ADALINE

- $a = pureline(W_i p + b_i) = W_{ip} + b_i$, where $W_i = [w_{i,1}, w_{i,1}, \ldots, w_{i,R}]^T$
- The decision boundary is obtain if $n = 0 \Rightarrow W_1^T p + b = 0$
- ADALINE can classify object which are linearly separable and in the case of figure 10.3 in two categories.

Deep Learning

Mean Square Error (MSE)

- Learning sequence:$\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_n, t_n\}$ where $\{p_i, t_i\}$ are respectively the i^{th} input to network and target output ($t \in R$).
- Let us note: $x = [w_1, b]^T$ and $z = [p, 1]^T \Rightarrow a = x^T z$
- The MSE: $F(x) = E[e^2] = E[(t-a)^2] = E[(t - x^T z)^2]$, where E is expected value operator.

$$\begin{aligned} F(x) &= E[t^2 - 2tx^T z + x^T zz^T x] \\ &= E[t^2] - 2x^T E[tz] + x^T E[zz^T]x \qquad (1) \\ F(x) &= c - 2x^T h + x^T Rx \qquad (2) \end{aligned}$$

where $c = E[t^2]$, $h = E[tz]$ and $R = E[zz^T]$.

- h gives the cross-correlation between the input vector and its associated target, while R is the input correlation matrix
- The function $F(x)$ is the performance index to be minimized w.r.t x and can be written in its general quadratic function form:

$$F(x) = c + \frac{1}{2}dx^T + \frac{1}{2}x^T Ax \qquad (3)$$

where $d = -2h$ and $A = 2R$.

Multilayer Neural Networks

Deep Learning-Mean Square Error (MSE)

- If A matrix is definite positive[a] then we can find a global minimum x^* of $F(x)$
- Calculation of the minimum point x^* based on stationary condition:

$$\nabla F(x) = 0 \Leftrightarrow \nabla\left(c + \frac{1}{2}dx^T + \frac{1}{2}x^T Ax\right) = d + Ax = -2h + 2Rx = 0 \Rightarrow x^* = R^{-1}h \quad (4)$$

- The existence of a unique solution depends only on the correlation matrix R. Therefore the characteristics of the input vectors determine whether or not a unique solution exists.
- If we could calculate the statistical quantities h and R, we could find the minimum point directly from equation (4).
- If we did not want to calculate the inverse of R, we could use the steepest descent algorithm, with the gradient $\nabla F(x) = -2h + 2Rx$ given in (4).
- In general it is not convenient to calculate h and R we can use an approximate steepest descent algorithm using an estimated gradient.

[a] In this case the Hessian matrix is twice the correlation matrix, and it can be shown that all correlation matrices are either positive definite or positive semidefinite which means that they can never have negative eigenvalues. If the correlation matrix has some zero eigenvalues, the performance index will either have a weak minimum or no minimum.

Deep Learning

LMS algorithme

- We can estimate the mean square error $F(x)$ by
$$\hat{F}(x) = (t(k) - a(k))^2 = e(k)^2 \tag{5}$$
where the expectation of the squared error has been replaced by the squared error at iteration k.
- At each iteration we have a gradient estimate of the form:
$$\hat{\nabla} F(x) = \nabla e(k)^2 \tag{6}$$
which is referred to as the stochastic gradient.
- When used in a gradient descent algorithm, it is referred to as 'on-line' or incremental learning, since the weights are updated as each input is presented to the network.
- The first R elements of $\nabla e(k)^2$ are derivatives with respect to the network weights, while the $(R+1)^{st}$ element is the derivative with respect to the bias:
$$[\nabla e^2(k)]_j = \frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}}, \quad \text{for} \quad j = 1, 2, \ldots, R, \tag{7}$$
$$[\nabla e^2(k)]_{R+1} = \frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}, \tag{8}$$

Deep Learning

LMS algorithme

- Let us consider the partial derivative terms at the ends of two previous equations.
$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial [t(k) - (w_1^T p(k) + b)]}{\partial w_{1,j}} = \frac{\partial [t(k) - \sum_{i=1}^{R} (w_{1,i} p_i(k) + b)]}{\partial w_{1,j}}, \tag{9}$$
where $p_i(k)$ is the *ith* element of the input vector at the *kth* iteration.
- This simplifies to
$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k). \tag{10}$$
- Similarly we can obtain the final element of the gradient
$$\frac{\partial e(k)}{\partial b} = -1. \tag{11}$$
- Recalling tha $z = [p\ 1]^T$ we can rewrite (6) as:
$$\hat{\nabla} F(x) = \nabla e(k)^2 = -2e(k)z(k). \tag{12}$$
the calculation of approximate gradient needs only to multiply the error times the input and can be used in the steepest descent algorithm:
$$x_{k+1} = x_k - \alpha \hat{\nabla} F(x) \mid_{x=x_k} = x_k + 2\alpha e(k)z(k), \tag{13}$$

Deep Learning

LMS algorithme

- We can distribute the update calculation of weights and biais values as:

$$\begin{aligned} w_1(k+1) &= w_1(k) + 2\alpha e(k) p(k) \\ b(k+1) &= b(k) + 2\alpha e(k) \end{aligned}$$

  which make up the least mean square (LMS) algorithm and also referred to as the delta rule or the Widrow-Hoff learning algorithm.
- In case where we have multiple outputs, and therefore multiple neurons[a], to update the *ith* row of the weight matrix use the update rule:

$$\begin{aligned} w_i(k+1) &= w_i(k) + 2\alpha e_i(k) p(k) \\ b_i(k+1) &= b_i(k) + 2\alpha e_i(k) \end{aligned}$$

- The LMS algorithm can be written conveniently in matrix notation:

$$\begin{aligned} W(k+1) &= W(k) + 2\alpha e(k) p^T(k) \\ b(k+1) &= b(k) + 2\alpha e(k) \end{aligned}$$

  where the error $e(k)$ and the bias $b(k)$ are now vectors.
- The maximum stable learning rate α for quadratic functions is $\alpha < 2/\lambda_{max}$, where λ_{max} is the largest eigenvalue of the Hessian matrix or A matrix in expression (3).

[a] see figure 10.1

Deep Learning

ADALINE Application: Adaptative Noise Cancellation

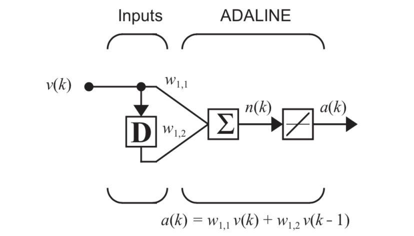


Figure 10.7 Adaptive Filter for Noise Cancellation

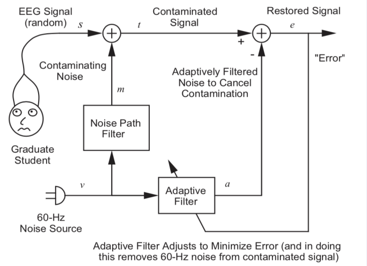

EEG Signal (random)
s
t
Contaminated Signal
Restored Signal
e
+
-
"Error"
Contaminating Noise
m
Adaptively Filtered Noise to Cancel Contamination
Graduate Student
Noise Path Filter
v
Adaptive Filter
a
60-Hz Noise Source
Adaptive Filter Adjusts to Minimize Error (and in doing this removes 60-Hz noise from contaminated signal)

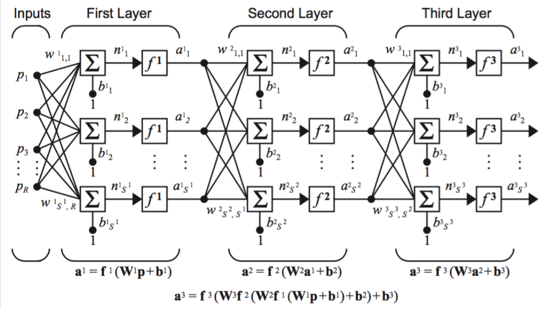
Figure 10.6 Noise Cancellation System

- Simulate nnd10nc.m

Deep Learning

Multi-Layer Perceptron Architecture

- The multilayer perceptron, trained by the back-propagation algorithm, is currently the most widely used neural network.



$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)$ $\mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2)$ $\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$

$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$

Figure 11.1 Three-Layer Network

Deep Learning

Backpropagation Algorithm

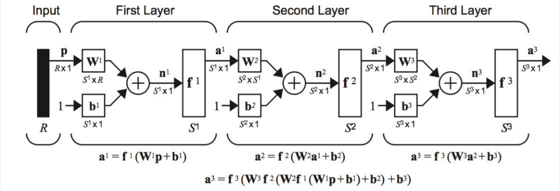


Figure 11.7 Three-Layer Network, Abbreviated Notation

- The equations that describe this operation at each layer $m = 0, 1, 2, \ldots, M-1$ is given by:

$$a^{m+1} = f^{m+1}(W^{m+1}a^m + b^{m+1}) \text{ for } m = 0, 1, \ldots, M-1 \tag{14}$$

- The neurons in the first layer receive external inputs:

$$a^0 = p \tag{15}$$

- The outputs of the neurons in the last layer are considered the network outputs:

$$a = a^M \tag{16}$$

Deep Learning

Back Propagation: Performance Index

- The backpropagation algorithm for multilayer networks is a generalization of the LMS algorithm and uses the same peformance index: Mean Square Error
- The algorithm is provided with a set of examples of proper network behavior:

$$\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_Q, t_Q\}, \tag{17}$$

  where p_q is an input to the network, and t_q is the corresponding target output.
- Learning consists to:

$$Min_{\mathbf{x}} F(\mathbf{x}) = E[e^2] = E[(t-a)^T(t-a)] \tag{18}$$

$$= [(t(k)-a(k))^T(t(k)-a(k))], k = 1, 2, \ldots, \quad Q \tag{19}$$

  where $\mathbf{x}$ is the matrix of network weights and biases.

Deep Learning

Back Propagation: Chain Rule

- Different optimization algorithms can be used to resolve the problem (18). If we apply the gradient algorithm we can write:

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \frac{\partial F}{\partial w_{i,j}^m}, \tag{20}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial F}{\partial b_i^m}, \tag{21}$$

  where α is the learning rate.
- From the multilayer NN given Figure 11.7 we can see that the performance index considered is a function of neurons outputs at each layer which are function of their inputs, weights and biases parameters.
- The following chained rule has to be applied in order to calculate the update values of weights and biases:

$$\frac{\partial F}{\partial w_{i,j}^m} = \frac{\partial F}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} \wedge \frac{\partial F}{\partial b_i^m} = \frac{\partial F}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} \tag{22}$$

- From the expression: $n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m$, the partial derivatives $\frac{\partial n_i^m}{\partial w_{i,j}^m}$ and $\frac{\partial n_i^m}{\partial b_i^m}$ in (22) can be easily calculated.

Deep Learning

Back Propagation: Sensitivity

- If we define the sensitivity of F to changes in the i^{th} element of the net input at layer m as :

$$s_i^m = \frac{\partial F}{\partial n_i^m}, \tag{23}$$

then the equation (22) can be simplified to:

$$\frac{\partial F}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} \wedge \frac{\partial F}{\partial b_i^m} = s_i^m, \tag{24}$$

- We can now express the approximate gradient descent algorithm as:

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1} \tag{25}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha s_i^m \tag{26}$$

and in matrix form:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T \tag{27}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m \tag{28}$$

$\mathbf{s^m} = \partial F / \partial \mathbf{n^m} = [\partial F / \partial \mathbf{n}_1^m, \partial F / \partial \mathbf{n}_2^m, \ldots, \partial F \partial / \mathbf{n}_{Sm}^m]^T$

Deep Learning

Back Propagation: Sensitivity

- It is necessary to compute the sensitivities $\mathbf{s}^m$, which requires another application of the chain rule describing the recurrence relationship in which the sensitivity at layer m is computed from the sensitivity at layer $m+1$.
- Let us give the Jacobian matrix:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots \frac{\partial n_2^{m+1}}{\partial n_{S^m}^m} \\ \vdots & \vdots & \vdots \\ \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_2^m} & \cdots \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix} \tag{29}$$

which can be obtained from:

$$\begin{aligned} \frac{\partial n_i^{m+1}}{\partial n_j^m} &= \frac{\partial \left(\sum_{l=1}^{S^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1} \right)}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial a_j^m}{\partial n_j^m} \\ &= w_{i,j}^{m+1} \frac{\partial f^m(n_j^m)}{\partial n_j^m} = w_{i,j}^{m+1} \dot{f}^m(n_j^m) \wedge \dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m} \end{aligned}$$

Deep Learning

Back Propagation: Sensitivity

- Therefore the Jacobian matrix can be written:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1}\dot{\mathbf{F}}^m(\mathbf{n}^m), \tag{30}$$

  where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}, \tag{31}$$

- We can now write out the recurrence relation for the sensitivity by the chain rule in matrix form:

$$\begin{aligned} \mathbf{s}^m &= \frac{\partial \mathbf{F}}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \times \frac{\partial \mathbf{F}}{\partial \mathbf{n}^{m+1}} = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \frac{\partial \mathbf{F}}{\partial \mathbf{n}^{m+1}} \\ &= \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1} \end{aligned} \tag{32}$$

- From (32) the sensitivities are propagated backward through the network from the last layer to the first layer as : $\mathbf{s}^M \rightarrow \mathbf{s}^{M-1} \rightarrow \dots \mathbf{s}^2 \rightarrow \mathbf{s}^1$

Deep Learning

Back Propagation: Sensitivity

- We need the starting point, $\mathbf{s}^M$, for the recurrence relation (32) This is obtained at the final layer:

$$s_i^M = \frac{\partial F}{\partial n_i^M} = \frac{\partial (\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})}{\partial n_i^M} = \frac{\partial \sum_{j=1}^{S^M}(t_j - a_j)^2}{\partial n_i^M} = -2(t_i - a_i)\frac{\partial a_i}{\partial n_i^M}$$

- Since $\frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M(n_i^M)}{\partial n_i^M} = \dot{f}(n_i^M) \Rightarrow s_i^M = -2(t_i - a_i)\dot{f}^M(n_i^M)$ and in matrix form:

$$\mathbf{s}^M = -2\dot{\mathbf{F}}(\mathbf{n}^M)(\mathbf{t}-\mathbf{a}). \tag{33}$$

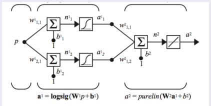
Deep Learning

Back Propagation: Summary

- Summary of the backpropagation algorithm:
  1. $\mathbf{a}^0 = \mathbf{p}$,
  2. $\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{\mathbf{m+1}}\mathbf{a}^m + \mathbf{b}^{m+1}), \quad m = 0, 1, \ldots, M-1,$
  3. $\mathbf{a} = \mathbf{a}^M$
  4. $\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a})$ and
     $\mathbf{s}^m = -2\dot{\mathbf{F}}^m(\mathbf{n})^m(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, \quad m = M-1, \ldots, 2, 1.$
  5. $\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T$ and $\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m$.

Example: Back Propagation

- Objective: Approximate the function $g(p) = 1 + sin(\frac{\pi}{4}p)$ for $-2 \leq p \leq 2$.
- Initial weights and biased values are chosen as small random values. $\mathbf{W}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}$ $\mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}$, $\mathbf{W}^2(0) = [0.09, -0.17], b^2(0) = [0.48]$



$w^1_{1,1}$
n^1_1
a^1_1
b^1_1
p
$w^1_{2,1}$
n^1_2
a^1_2
b^1_2
1
$w^2_{1,1}$
$w^2_{1,2}$
n^2
a^2
b^2
$\mathbf{a}^1 = \mathbf{logsig}(\mathbf{W}^1 p + \mathbf{b}^1)$
$a^2 = purelin(\mathbf{W}^2\mathbf{a}^1 + b^2)$

Figure 11.8 Example Function Approximation Network

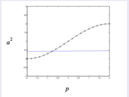


Figure 11.9 Initial Network Response

- We sample the function at 21 points in the range $[-2, 2]$ at equally spaced intervals of 0.2 and $p = a^0 = 1$
- The training points can be presented in any order, but they are often chosen randomly.
- The output of the first layer is then:

$$a^1 = f^1(W^1 a^0 + b^1) = logsig(\begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}[1] + \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}) = logsig(\begin{bmatrix} -0.75 \\ -0.54 \end{bmatrix}) = \begin{bmatrix} \frac{1}{1+e^{0.75}} \\ \frac{1}{1+e^{0.54}} \end{bmatrix} = \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix}$$

Deep Learning

Example: Back Propagation

- The second layer output is: $a^2 = f^2(W^2a^1 + b^2) = pureline(\begin{bmatrix} 0.09 & -0.17 \end{bmatrix} \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix} + \begin{bmatrix} 0.48 \end{bmatrix}) = \begin{bmatrix} 0.446 \end{bmatrix}$
- The error would then be: $e = t - a = \{1 + sin(\frac{\pi}{4}p)\} - a^2 = \{1 + sin(\frac{\pi}{4}p)\} - 0.446 = 1.261$
- The next stage of the algorithm is to backpropagate the sensitivities which requires a prior calculation of transfer functions derivatives, $\dot{f}^1(n)$, $\dot{f}^2(n)$ for the layers n^1 and n^2.

$$\dot{f}^1(n) = \frac{d}{dn}(\frac{1}{1+e^{-n}}) = \frac{e^{-n}}{(1+e^{-n})^2} = (1 - \frac{1}{1+e^{-n}})(\frac{1}{1+e^{-n}}) = (1-a^1)(a^1) \quad \text{and} \quad \dot{f}(n^2) = 1.$$

- Operate the backpropagation with starting point the second layer:

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}((\mathbf{n^2})(\mathbf{t} - \mathbf{a}) = -2[\dot{f}(n^2)](1.261) = -2[1](1.261) = -2.522$$

.

$$\begin{aligned} \mathbf{s}^1 &= \dot{\mathbf{F}}(\mathbf{n^1})(\mathbf{W^2})^{\mathbf{T}}\mathbf{s^2} = \begin{bmatrix} (1-a_1^1)(a_1^1) & 0 \\ 0 & (1-a_2^1)(a_2^1) \end{bmatrix} \begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix} \begin{bmatrix} -2.522 \end{bmatrix} \\ &= \begin{bmatrix} (1-0.321)(0.321) & 0 \\ 0 & (1-0.368)(0.368) \end{bmatrix} \begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix} \begin{bmatrix} -2.522 \end{bmatrix} = \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} \end{aligned} \tag{34}$$

.

Deep Learning

Example: Back Propagation

- The final stage of the algorithm is to update the weights and we chose a learning rate $\alpha = 0.1$.
- $\mathbf{W}^2(1) = \mathbf{W}^2(0) - \alpha \mathbf{s}^2(\mathbf{a}^1)^T = [0.09 \;\; -0.17] - 0.1[-2.522][0.321 \;\; 0.368] = [0.171 \;\; -0.0772]$, $\mathbf{b}^2(1) = \mathbf{b}^2(0) - \alpha \mathbf{s}^2 = [0.48] - 0.1[-2.522] = [0.732]$,
  $\mathbf{W}^1(1) = \mathbf{W}^1(0) - \alpha \mathbf{s}^1(\mathbf{a}^0)^T = \begin{bmatrix}-0.27\\-0.41\end{bmatrix} - 0.1 \begin{bmatrix}-0.0495\\0.0997\end{bmatrix}[1] = \begin{bmatrix}-0.265\\-0.420\end{bmatrix}$
  $\mathbf{b}^1(1) = \mathbf{b}^1(0) - \alpha \mathbf{s}^1 = \begin{bmatrix}-0.48\\-0.13\end{bmatrix} - 0.1 \begin{bmatrix}-0.0495\\0.0997\end{bmatrix} = \begin{bmatrix}-0.475\\-0.140\end{bmatrix}$
- This completes the first iteration of the backpropagation algorithm and we proceed to randomly choose another input from the training set and perform another iteration of the algorithm until the difference between the network response and the target function reaches some acceptable level[a].
- Use Matlab file nnd11bc.m to simulate the above first iteration of neural network weight's calculation.
- Use Matlab file nnd11fa.m to simulate a function approximation using Back-Prapagation algorthm.

[a] Note that this will generally take many passes through the entire training set

Deep Learning

Dynamic Networks

- Neural networks can be classified into static and dynamic categories.
- The multilayer network that we have already discussed are static network means that the output can be calculated directly from the input through feedforward connections.
- In dynamic networks, the output depends not only on the current input to the network, but also on the current or previous inputs, outputs or states of the network. They contain delays [a], operate on a sequence of inputs and have memory[b].
- Dynamic networks can be trained to learn sequential or time-varying patterns such as a dynamic system with applications in such diverse areas as control of dynamic systems, prediction in financial markets, channel equalization in communication systems, phase detection in power systems, sorting, fault detection, speech recognition, learning of grammars in natural languages, and even the prediction of protein structure in genetics.

[a] or integrators, for continuous-time networks

[b] Their response at any given time will depend not only on the current input, but on the history of the input sequence

Multilayer Neural Networks

Dynamic Networks

- Dynamic networks can be trained using the standard optimization methods however, the gradients and Jacobians that are required for these methods cannot be computed using the standard backpropagation algorithm.
- There are two general approaches to gradient and Jacobian calculations in dynamic networks backpropagation-through- time (BPTT) [Werb90] and real-time recurrent learning (RTRL) [WiZi89].
- In the BPTT algorithm, the network response is computed for all time points, and then the gradient is computed by starting at the last time point and working backward in time. This algorithm is efficient for the gradient calculation, but it is difficult to implement on-line, because the algorithm works backward in time from the last time step.
- In the RTRL algorithm, the gradient can be computed at the same time as the network response, since it is computed by starting at the first time point, and then working forward through time. RTRL requires more calculations than BPTT for calculating the gradient, but RTRL allows a convenient framework for on-line implementation.

Deep Learning

Layered Digital Dynamic Networks

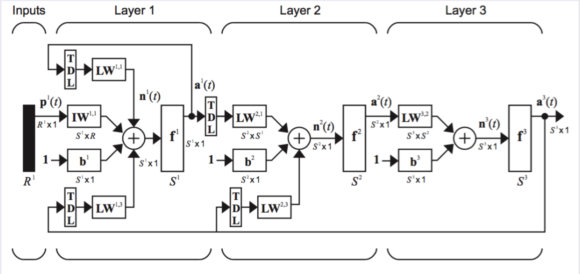


Figure 14.1 Example Dynamic Network

- The general equations for the computation of the net input $n^m(t)$ for layer m of an *LDDN* are:

$$
\begin{aligned}
n^m(t) \quad &= \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} LW^{m,l}(d)a^l(t-d) \\
&+ \sum_{l \in I_m} \sum_{d \in DI_{m,l}} IW^{m,l} p^l(t-d) + b^m \qquad (35)
\end{aligned}
$$

- $p^l(t)$: is the l^{th} input vector to the network at time t

Deep Learning

Layered Digital Dynamic Networks

- $IW^{m,l}$ is the input weight between input l and layer m,
- $LW m, l$ is the layer weight between layer l and layer m,
- b^m is the bias vector for layer m.

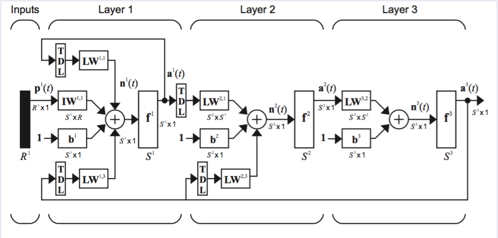


Figure 14.1 Example Dynamic Network

- $DL_{m,l}$ is the set of all delays in the tapped delay line between Layer l and Layer m,
- $DI_{m,l}$ is the set of all delays in the tapped delay line between Input l and Layer m,
- I_m is the set of indices of input vectors that connect to layer m,
- L_m^f is the set of indices of layers that directly connect forward to layer m.
- The output of layer m is then computed as

$$a^m(t) = f^m(n^m(t)). \tag{36}$$

Deep Learning

Layered Digital Dynamic Networks

- Within static multilayer networks, the layers were connected to each other in numerical order. In other words, Layer 1 was connected to Layer 2, which was connected to Layer 3, etc. Within the LDDN framework, any layer can connect to any other layer, even to itself.
- The order in which the layer outputs given in (35) must be computed to obtain the correct network output is called the simulation order.
- In order to backpropagate the derivatives for the gradient calculations, we must proceed in the opposite order, which is called the backpropagation order.
- In *Figure 14.1*, the standard numerical order, $1-2-3$, is the simulation order, and the backpropagation order is $3-2-1$.

Deep Learning

Layered Digital Dynamic Networks

- As with the multilayer network, the fundamental unit of the *LDDN* is the layer. Each layer in the *LDDN* is made up of five components:
  1. a set of weight matrices that come into that layer (which may connect from other layers or from external inputs),
  2. any tapped delay lines (represented by $DL_{m,l}$ or $DI_{m,l}$) that appear at the input of a set of weight matrices,
  3. a bias vector,
  4. a summing junction, and
  5. a transfer function.
- The weights and biases have two different effects on the network output:
  1. The *first* is the direct effect, which can be calculated using the *standard backpropagation algorithm*,
  2. The *second* is an indirect effect, since some of the inputs to the network are previous outputs, which are also functions of the weights and biases.

Deep Learning

Example: Layered Digital Dynamic Networks

- The Adaline NN given in *Figure 14.2* has a *TDL* on the input, with $DL_{1,1} = \{0, 1, 2\}$
- To demonstrate the operation of this network, we will apply a square wave as input, and we will set all of the weight values equal to $1/3$:

  $iw_{1,1}(0) = 1/3; iw_{1,1}(1) = 1/3; iw_{1,1}(2) = 1/3;$

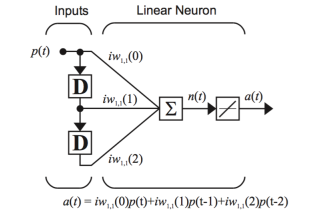


Figure 14.2 Example Feedforward Dynamic Network

The network response is calculated from:

$$\begin{aligned} a(t) &= n(t) = \sum_{d=0}^{2} IW(d)p(t-d) \\ &= n(t) = iw_{1,1}(0)p(t) + iw_{1,1}(1)p(t-1) + iw_{1,1}(2)p(t-2) \end{aligned} \tag{37}$$

where we have left off the superscripts on the weight and the input, since there is only one input and only one layer.

Deep Learning

Example: Layered Digital Dynamic Networks

- The response of the network is shown in Figure 14.3 where the open circles represent the square-wave input signal $p(t)$ and the dots represent the network response $a(t)$.

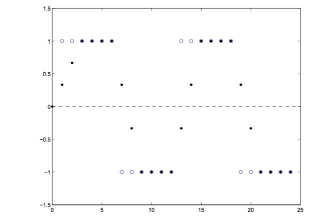


Figure 14.3 Response of ADALINE Filter Network

- For this dynamic network, the response at any time point depends on the previous three input values. If the input is constant, the output will become constant after three time steps(FIR filter).

Deep Learning

Example: Layered Digital Dynamic Networks

- Consider another simple linear dynamic network, but one that has a recurrent connection.
- The equation of operation of the network is:

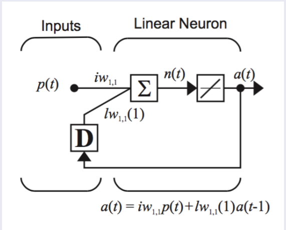


Figure 14.4 Recurrent Linear Neuron

$$
\begin{aligned}
a^1(t) &= n^1(t) = LW^{1,1}a^1(t) + IW^{1,1}(0)p(t) \\
&= n^1(t) = iw_{1,1}(0)p(t) + lw_{1,1}(1)a(t-1) \\
& \quad iw_{1,1}(0) = 1/2, \;\; lw_{1,1} = 1/2
\end{aligned} \tag{38}
$$

where we have left off the superscripts on the weight and the input, since there is only one input and only one layer.

Deep Learning

Example: Layered Digital Dynamic Networks

- The response of this network to the square wave input is shown in *Figure 14.5*. The network responds exponentially to a constant change in the input sequence.
- Unlike the FIR filter network of Figure 14.2, the exact response of the network at any given time is a function of the infinite history of inputs to the network.

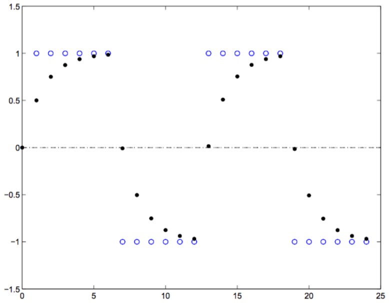


Figure 14.5 Recurrent Neuron Response

- Because dynamic systems are more complex than static functions, the training process for dynamic networks will be more challenging than static network training.

Deep Learning

Principles of Dynamic Learning

- Consider again the recurrent network of *Figure 14.4* and suppose that we want to train the network using steepest descent where Q is the training horizon.
- The first step is to compute the gradient of the performance function given by

$$F(x) = \sum_{i=1}^{Q} e^2(t) = \sum_{i=1}^{Q} (t(t) - a(t))^2 \qquad (39)$$

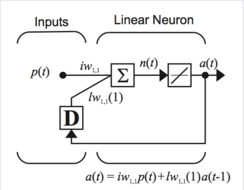


Figure 14.4 Recurrent Linear Neuron

- The two elements of the gradient will be:

$$\frac{\partial F(x)}{\partial lw_{1,1}(1)} = \sum_{i=1}^{Q} \frac{\partial e^2(t)}{\partial lw_{1,1}(1)} = -2\sum_{i=1}^{Q} e(t)\frac{\partial a(t)}{\partial lw_{1,1}(1)} \qquad (40)$$

$$\frac{\partial F(x)}{\partial iw_{1,1}} = \sum_{i=1}^{Q} \frac{\partial e^2(t)}{\partial iw_{1,1}} = -2\sum_{i=1}^{Q} e(t)\frac{\partial a(t)}{\partial iw_{1,1}} \qquad (41)$$

- The key terms in these two equations are the derivatives of the network output with respect to the weights:

$$\frac{\partial a(t)}{\partial lw_{1,1}(1)} \quad \text{and} \quad \frac{\partial a(t)}{\partial iw_{1,1}} \qquad (42)$$

Deep Learning

Principles of Dynamic Learning

- For static network they would correspond to $a(t-1)$ and $p(t)$, respectively and for *recurrent networks*, the weights have two effects on the network output.
- The equation of operation of the network is:

$$a(t) = lw_{1,1}a(t-1) + iw_{1,1}p(t) \tag{43}$$

- We can compute the terms in equation (42) by taking the derivatives of (43):

$$\frac{\partial a(t)}{\partial lw_{1,1}(1)} = a(t-1) + lw_{1,1}(1)\frac{\partial a(t-1)}{\partial lw_{1,1}(1)} \tag{44}$$

$$\frac{\partial a(t)}{\partial iw_{1,1}} = p(t) + lw_{1,1}(1)\frac{\partial a(t-1)}{\partial iw_{1,1}} \tag{45}$$

- The *first term* in each of these equations represents the direct effect that each weight has on the network output and the *second term* represents the indirect effect or the *feedback effect*.
- Note that unlike the gradient computation for static networks, the derivative at each time point depends on the derivative at previous time points[a].

[a] or at future time points, as we will see later

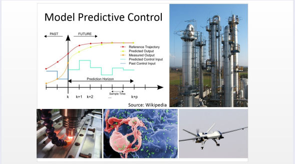
Deep Learning

Neural Network Controller

- There are typically two steps involved when using neural networks for control: System Identification and Control Design.
- In the system identification stage, we develop a neural network model of the plant that we want to control.
- In the control design stage, we use the neural network plant model to design (or train) the controller.
- Three control architecture have been successfully applied: Model Predictive Control, NARMA-L2 Control and Model Reference Control.
- In each of the these three control architectures, the system identification stage is necessary and identical.

Model Predictive Control

- The neural network predictive controller uses a neural network model of a nonlinear plant to predict future plant performance.
- The controller then calculates the control input that will optimize plant performance over a specified future time horizon.



Deep Learning

Model Predictive Control: System Identification

- The first stage of model predictive control (as well as the other two control architectures discussed here) is to train a neural network to represent the forward dynamics of the plant.
- The prediction error between the plant output and the neural network output is used as the neural network training signal.

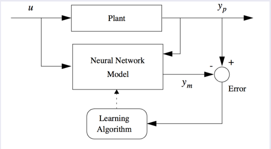


Figure 13 Plant Identification

Deep Learning

Model Predictive Control: System Identification

- One standard model that has been used for nonlinear identification is the Nonlinear Autoregressive-Moving Average (NARMA) model([a]):

  $$y(k+d) = h(y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-m+1)) \quad (46)$$

  where $u(k)$ is the system input, $y(k)$ is the system output and d is the delay
- The equation for the plant model implemented by a Neural Network can be given by:

  $$y(k+h) = \hat{h}(y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-m+1) \quad (47)$$

  where **x** is the vector containing all network weights and biases.
- $IW_{i,j}$ is a weight matrix from input number j to layer number i and $LW_{i,j}$ is a weight matrix from layer number j to layer number i.

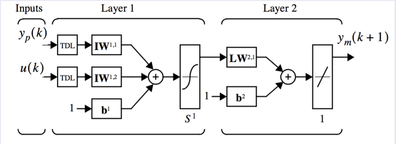


Figure 14 Neural Network Plant Model

[a] We will use a delay of 1 for the predictive controller

Deep Learning

Model Predictive Control: System Identification

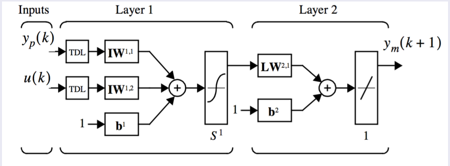


Figure 14 Neural Network Plant Model

- Although there are delays in this network, they occur only at the network input, and the network contains no feedback loops. So, the neural network plant model can be trained using the backpropagation methods for feedforward networks if we consider its input as $(n_y + n_u)$ dimensional vector of previous plant outputs and inputs which has to be covered adequately by the training data.
- It is important that the training data cover the entire range of plant operation, because nonlinear neural networks do not extrapolate accurately. The input to this network is an $(ny + nu)$ dimensional vector of previous plant outputs and inputs.

Deep Learning

Model Predictive Control

- The neural network model predicts the plant response over a specified time horizon and used by a numerical optimization program to determine the control signal that minimizes the following performance criterion over the specified horizon:

$$J = \sum_{j=N_1}^{N_2} (y_r(k+j) - y_m(k+j))^2 + \rho \sum_{j=1}^{N_u} (u^{'}(k+j-1) - u^{'}(k+j-2))^2 \qquad (48)$$

- N_1, N_2, N_u define the horizon over which the tracking error and the control increments are evaluated.
- The $u^{'}, y_r, y_m$ are (tentative) control signal, desired response and network model response respectively.
- The ρ value determines the contribution that the sum of the squares of the control increments has on the performance index.

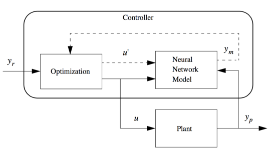


Figure 15 Neural Network Predictive Control

Deep Learning

Model Predictive Control: Application to Magnetic Levitation System

- The objective is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction. Its dynamic model is given by:

$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)sgn(i(t))}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt} \qquad (49)$$

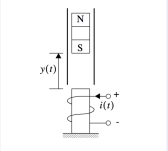


Figure 16 Magnetic Levitation System

- $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, M is the mass of the magnet, and g is the gravitational constant.
- The parameter β is a viscous friction coefficient that is determined by the material in which the magnet moves, and α is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet.
- For our simulations, the current is allowed to range from 0 to 4 *amps*, and the sampling interval for the controller is 0.01 *seconds*. The parameter values are set to $\beta = 12$, $\alpha = 15$, $g = 9.8$ and $M = 3$.
- We need to insure that the plant input is sufficiently exciting. For nonlinear black box identification, we also need to be sure that the system inputs and outputs cover the operating range for which the controller will be applied.

Deep Learning

Model Predictive Control: Application to Magnetic Levitation System

- For our applications, we typically collect training data while applying random inputs which consist of a series of pulse of random amplitude and duration.
- We found that open loop time response of magnetic levitation system is 4.5 seconds and we specified a pulse width range of $0.01 < \tau < 5$.
- The neural network plant model used three delayed values of current ($m = 3$) and three delayed values of magnet position ($n = 3$) as input to the network, and 10 neurons were used in the hidden layer.

- After training the network with the data set shown in *Figure 17*, the resulting neural network predictive control system was unstable therefore, we changed the range of the input pulse widths to $0.01 < \tau < 1$, as shown in *Figure 18*.
- After training the network using this data set, the resulting predictive control system was stable, although it resulted in large steady- state errors.

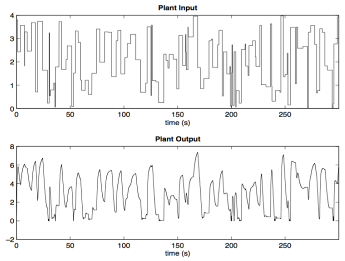


Figure 17 Training data with a long pulse width.

Deep Learning

Model Predictive Control: Application to Magnetic Levitation System

- After training the network using this data set, the resulting predictive control system was stable, although it resulted in large steady- state errors.

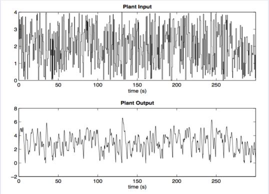


Figure 18 Training data with a short pulse width.

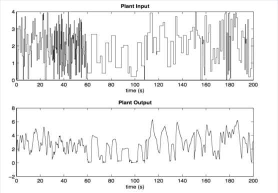


Figure 19 Training data with mixed pulse width.

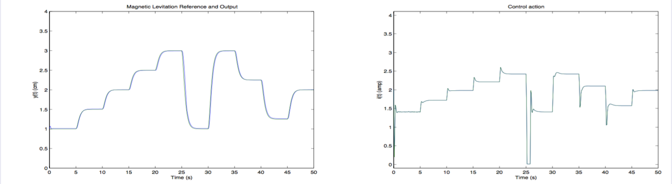


Figure 20 MagLev response and control action using the Predictive Controller

Deep Learning

Model Reference Control

- This architecture uses two neural networks: a controller network and a plant model network, as shown in *Figure 25*. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.
- Model reference control architecture requires that a separate neural network controller be trained, in addition to the neural network plant model.
- The controller training is computationally expensive, since it requires the use of dynamic backpropagation.

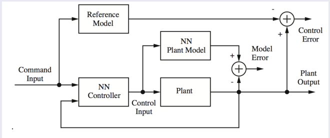
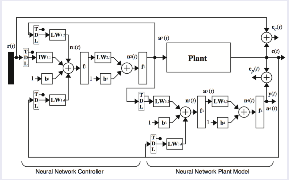


Figure 25 Model Reference Control Architecture



- This architecture uses two neural networks: a controller network and a plant model network, as shown in figure. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.
- The data used to train the model reference controller is generated while applying a random reference signal which consists of a series of pulses of random amplitude and duration. This data can be generated without running the plant, but using the neural network model output in place of the plant output.

Deep Learning

Model Reference Control: Application to Robot Arm

- The equation of motion for the arm is:

$$\frac{d^2\phi}{dt^2} = -10 sin\phi - 2\frac{d\phi}{dt} + u \quad (50)$$

where ϕ is the angle of the arm, and u is the torque supplied by the DC motor.

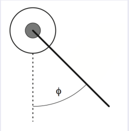


Figure 27 Single-link robot arm

- To identify the system, we used input pulses with intervals between 0.1 and 2 seconds, amplitudes between -15 and $+15$ *[Nm]* and sampling period is equal to 0.05 seconds. The neural network plant model used two delayed values of torque ($m = 2$) and two delayed values of arm position ($n = 2$) as input to the network, and 10 neurons were used in the hidden layer (a $5-10-1$ network). The reference model is given in (51).

$$\frac{d^2 y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r \quad (51)$$

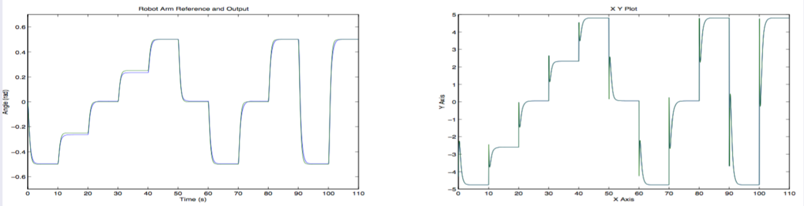


Figure 28 Robot arm response and control action for the Model Reference Controller

Annexes: Non-constrained Optimization

Necessary Conditions of Optimality

- Consider the optimization problem: $min_{x \in \mathscr{R}^n} f(x)$
- **Theorem** Let $f(x)$ be C^2 ($f(x)$ and its derivative are continuously differentiable functions). If x^* is a local minimum of $f(x)$, the $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is **positive semi-definite (PSD)**
- Exercise: Prove this theorem
- Example:

$$f(x) = \frac{1}{2}x_1^2 + x_1 x_2 + 2x_2^2 - 4x_1 - 4x_2 - x_2^3$$

$$\nabla^T f(x) = (x_1 + x_2 - 4, x_1 + 4x_2 - 4 - 3x_2^2)$$

. Two candidates $x^* = \{(4,0),(3,1)\}$ $\nabla^2 f(x) = \begin{bmatrix} 1 & 1 \\ 1 & 4 - 6x_2 \end{bmatrix}$

$\nabla^2 f((4,0)) = \begin{bmatrix} 1 & 1 \\ 1 & 4 \end{bmatrix}$ is **PSD**;

$\nabla^2 f((3,1)) = \begin{bmatrix} 1 & 1 \\ 1 & -2 \end{bmatrix}$ which is indefinite matrix. So (4,0) is the only candidate for local minimum

Annexes: Non-constrained Optimization

Sufficient Conditions of Optimality

- Consider the optimization problem:

$$min_{x \in \mathscr{R}^n} f(x)$$

- **Theorem** Let $f(x)$ be C^2 (twice differentiable) function. If $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is **positive semi-definite (PSD)** in the ball $B(x^*, \varepsilon)$, then x^* is a local minimum.
- Exercise: Prove this theorem
- **Example:** (from the precedent slide): At $x^* = (4,0)$, $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is **PSD** for $x \in B(x^*, \varepsilon)$
- $$f(x) = x_1^3 + x_2^2 \Rightarrow \nabla^T f(x) = (3x_1^2, 2x_2) \wedge x^* = (0,0)$$
- $$\nabla^2 f(x) = \left[\begin{array}{cc} 6x_1 & 0 \\ 0 & 2 \end{array} \right]$$

  is not PSD in $B(0, \varepsilon)$, because $f(-\varepsilon, 0) = -\varepsilon^3 < 0 = f(x^*)$

Annexes: Non-constrained optimization

Characterization of convex functions

- **Theorem** Let $f(x)$ be continuously differentiable. Then $f(x)$ is convex if and only if
$$\nabla^T f(x)(\bar{x} - x) \leq f(\bar{x}) - f(x) \tag{52}$$
- Exercise: Prove the theorem
- **Theorem** Let $f(x)$ be a continuously differentiable convex function. Then x is a minimum of $f(x)$ if and only if
$$\nabla f(x^*) = 0$$
- Exercise: Prove the theorem

Annexes: Non-constrained optimization

Descent Directions: Interesting Observation

- If $f(x)$ is differentiable at $\bar{x} \neq x^*$ then $\exists d : \nabla^T f(\bar{x}) d < 0 \Rightarrow \forall \lambda > 0$ sufficiently small $f(\bar{x} + \lambda d) < f(\bar{x})$
- d is a descent direction
- Proof:
$$f(\bar{x} + \lambda d) = f(\bar{x}) + \lambda \nabla^T f(\bar{x}) d + \lambda \|d\| R(\bar{x}, \lambda d)$$
with $R(\bar{x}, \lambda d)_{(\lambda \to 0)} \to 0$
$$f(\bar{x} + \lambda d) - f(\bar{x}) = \lambda \nabla^T f(\bar{x}) d + \lambda \|d\| R(\bar{x}, \lambda d)$$
If $\nabla^T f(\bar{x}) d < 0$ and λ sufficiently small ($R(\bar{x}, \lambda d)_{(\lambda \to 0)} \to 0$)
$f(\bar{x} + \lambda d) < f(\bar{x})$ QED

Annexes: Non-constrained optimization

Gradient Methods: Motivation

- Decrease $f(x)$ until $\nabla f(x^*) = 0$
- $f(\bar{x} + \lambda d) \approx f(\bar{x}) + \lambda \nabla^T f(\bar{x}) d$
- if $\nabla^T f(\bar{x}) d < 0$, then for small $\lambda > 0 \Rightarrow f(\bar{x} + \lambda d) < f(\bar{x})$

Algorithm

1. Choose step length $\lambda^k > 0$
2. Choose descending direction d^k such that $\nabla^T f(x^k) d < 0$
3. $x^{k+1} = x^k + \lambda^k d^k$
4. if $\|\nabla f(x^k)\| > \varepsilon_f \vee \|x^{k+1} - x^k\| > \varepsilon_x$ go to 1
5. Print x^{k+1} and stop calculation
6. Another recurrent calculation: $x^{k+1} = x^k - \lambda^k D^k \nabla f(x^k)$, D^k is a positive definite symmetric matrix

Annexes: Non-constrained optimization

Steepest Descend Method: Algorithm

1. Step 0 given x^0 set $k = 0$
2. Step 1 $d^k = -\nabla f(x^k)$. If $\|d^k\| \leq \varepsilon$ then stop.
3. Step 2 Solve $min_\lambda h(\lambda) := f(x^k + \lambda d^k)$ for the step-length λ_k, perhaps chosen by an exact or inexact line-search.
4. Step 3 $x^{k+1} = x^k + \lambda^k d^k, k \leftarrow k+1$ Go To Step 1

Annexes: Non-constrained optimization

Steepest Descend Method: Example

- $f(x_1, x_2) = 5x_1^2 + x_2^2 + 4x_1x_2 - 14x_1 - 6x_2 + 20$
- $x^* = (x_1^*, x_2^*)^T = (1, 1)^T$, $f(x^*) = 10$
- Given x^k,

$$d^k = -\nabla f(x_1^k, x_2^k) = \begin{bmatrix} -10x_1^k - 4x_2^k + 14 \\ -2x_2^k - 4x_1^k + 6 \end{bmatrix} = \begin{bmatrix} d_1^k \\ d_2^k \end{bmatrix}$$

-

$$\begin{aligned} h(\lambda) &= f(x^k + \lambda d^k) && (53) \\ &= 5(x_1^k + \lambda d_1^k)^2 + (x_2^k + \lambda d_2^k)^2 + 4(x_1^k + \lambda d_1^k)(x_2^k + \lambda d_2^k) && (54) \\ &\quad -14(x_1^k + \lambda d_1^k) - 6(x_2^k + \lambda d_2^k) + 20 && (55) \\ & && (56) \end{aligned}$$

Annexes: Non-constrained optimization

Steepest Descend Method: Example

- $\lambda^k = \frac{(d_1^k)^2+(d_2^k)^2}{2(5(d_1^k)^2+(d_2^k)^2+4d_1^k d_2^k)}$
- start at $x = (0, 10)$
- $\varepsilon = 10^{-6}$

Example: Important Properties

- $f(x_{k+1}) < f(x_k) < \ldots < f(x_0)$ (because d_k are descent directions)
- Under reasonable assumptions of $f(x)$, the sequence $x_0, x_1, \ldots$, will have at least one cluster point $\bar{x}$
- Every cluster point $\bar{x}$ will satisfy $\nabla f(\bar{x}) = 0$
- Implication: If $f(x)$ is a convex function, $\bar{x}$ will be an optimal solution
- **Theorem:** If $f(x) : R^n \to R$ is continuously differentiable on $\mathscr{F} = \{x \in R^n : f(x) \leq f(x_0)\}$ closed, bounded set. Every cluster point $\bar{x}$ of $\{x_k\}$satisfies $\nabla f(\bar{x}) = 0$.

Annexes: Non-constrained optimization

Newton Method: Motivation

- Consider the optimization problem without constraints: $\min_x \quad f(x)$
- The Taylor expression of $f(x)$ around $\bar{x}$

$$\Rightarrow f(x) \approx g(x) = f(\bar{x}) + \nabla^T f(\bar{x})(x-\bar{x}) + \frac{1}{2}(x-\bar{x})^T \nabla^2 f(\bar{x})(x-\bar{x})$$

- Instead of $min \quad f(x)$, solve $min \quad g(x) \Rightarrow \nabla g(x) = 0$
- $\Rightarrow \nabla f(\bar{x}) + \nabla^2 f(\bar{x})(x-\bar{x}) = 0 \Rightarrow x-\bar{x} = -(\nabla^2 f(\bar{x}))^{-1} \nabla f(\bar{x})$
- The direction $d = -(\nabla^2 f(\bar{x}))^{-1} \nabla f(\bar{x})$ is the Newton direction
- Newton Algorithm
  1. Step 0: Given $x0, \lambda$, set $k=0$
  2. Step 1: $d^k = -(\nabla^2 f(x^k))^{-1} \nabla f(x^k)$, if $\|d^k\| \leq \varepsilon$ Then stop
  3. Step 2: $x^{k+1} = x^k + \lambda d^k, k \leftarrow k+1$, Go to Step 1

Annexes: Non-constrained optimization

Newton Method: Example

- $f(x) = 7x - logx$, $x^* = \frac{1}{7} = 0.14857143$
- $f^{'}(x) = 7 - \frac{1}{x}$, $f^{''}(x) = \frac{1}{x^2}$
- $\Rightarrow d = -(f^{''}(x))^{-1} f^{'}(x) = -(\frac{1}{x^2})^{-1}(7 - \frac{1}{x}) = x - 7x^2$
- $x^{k+1} = x^k + (x^k - 7(x^k)^2) = 2x^k - 7(x^k)^2$
- Test for $x^0 = 1$, $x^0 = 0$, $x^0 = 0.1$, $x^0 = 0.01$