Neural Networks - Multilayer Perceptrons

Enrique Romero

Computational Intelligence
Master in Artificial Intelligence

Soft Computing Group
Computer Science Department
Universitat Politècnica de Catalunya, Barcelona, Spain

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons

Outline

1. Introduction
    - Definition
    - Biological Motivation
    - A Bit of History
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons

Outline

1. Introduction
   - Definition
   - Biological Motivation
   - A Bit of History
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons

Definition

Feed-forward Neural Networks are neural systems defined by:

- A set of interconnected **units** (neurons)
- A set of **directed connections** (synapses or weights) among the units **without loops**

Each unit

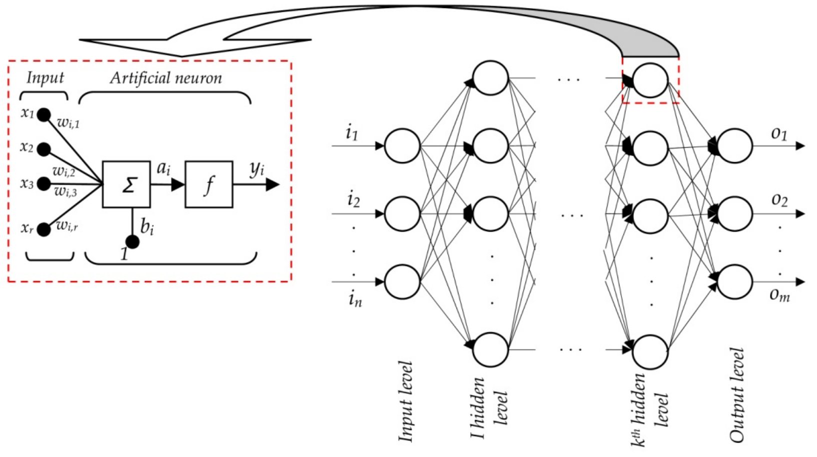
- Receives information of the units with connections **to** this unit
- Sends information to the units with connections **from** this unit

Typically,

- There are a number of **input units**, where the process starts
- Some of the units are **output units**, with the network results
- The rest of units are called **hidden** units
- Units are usually structured by **layers**

A Typical Example

A typical architecture of a Multilayer Perceptron



Different Types of Computations

The computation made by every unit depends on the type of neural networks we are working with, and usually consists in two steps:

- An **aggregation function**, that combines the information received by the unit with the weights (converting multidimensional information into a scalar)
- An **activation function**, that transforms (usually in a non-linear way) the output of the aggregation function

The output of the aggregation function is named **net-input** or **pre-activation**

The output of the activation function (the output of the unit) is named the **net-output** or **activation** of the unit

Input units are fed with the **data** (one example at a time)

Different Types of Computations

Examples of aggregation functions:

- For Multilayer Perceptrons (MLPs) and Recurrent Neural Networks, the inner product between the weights and the activation of the previous units:

$$\mathrm{act}_i = \boldsymbol{\omega}_i \cdot \mathrm{act\_to}_i = \sum_j \omega_{ij}\ \mathrm{act\_to}_{i_j}$$

- For Radial-basis Function Networks, the squared Euclidean distance between the weights and the activation of the previous units:

$$\mathrm{act}_i = ||\boldsymbol{\omega}_i - \mathrm{act\_to}_i||^2 = \sum_j \left(\omega_{ij} - \mathrm{act\_to}_{i_j}\right)^2$$

- For Convolutional Neural Networks, the convolution operation

In this lecture we will focus on MLPs, but similar ideas apply for other types of neural networks

Different Types of Computations

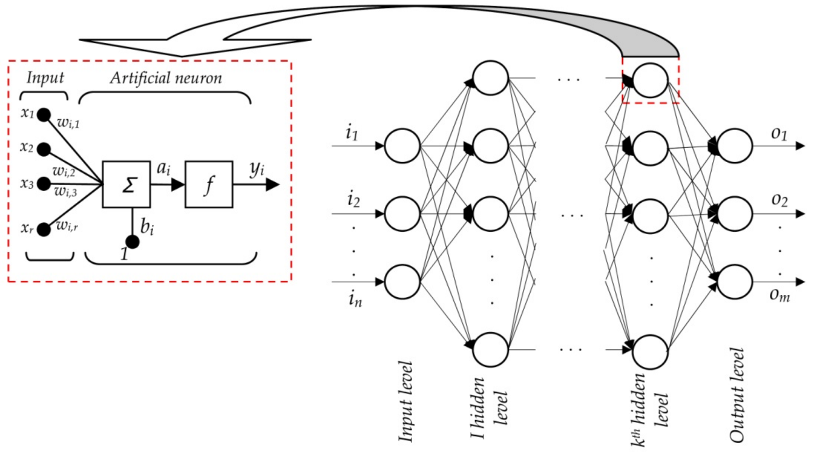
Examples of activation functions:

- Identity function: $f(z) = z$
- Logistic function: $f(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic tangent function: $f(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- Gaussian function: $f(z) = e^{-z/\sigma}$
- Rectifier linear function: $f(z) = \max(0, z)$
- Softplus function: $f(z) = \log(1 + e^{z})$
- Softmax function (output units): $f(z)_c = \frac{e^{z_c}}{\sum_k e^{z_k}}$
- ... and many more

In many cases, the aggregation and the activation function are considered a single function, since every activation function is usually combined with a unique aggregation function

A Typical Example (revisited)

How is the computation performed?



Outline

1. Introduction
   - Definition
   - Biological Motivation
   - A Bit of History
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons

Biological Motivation

The term "Artificial Neural Network" was originally motivated by the idea of modeling networks of real neurons in the brain

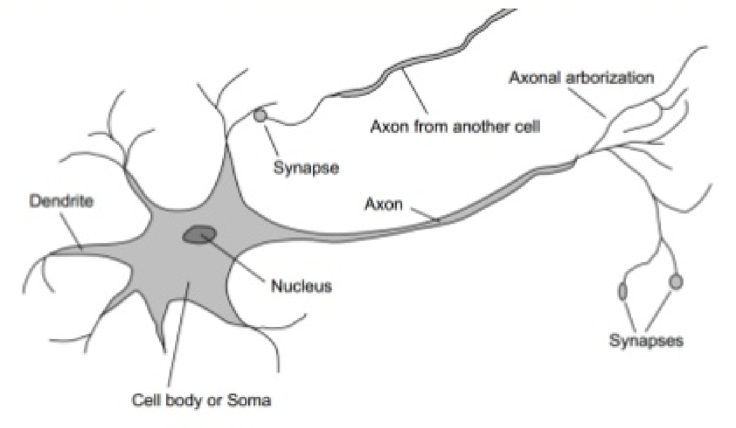
Nowadays, we know that the artificial neurons used to construct artificial neural networks are very primitive approximations to those found in the brain

Therefore, also the neural network architectures used are extremely simple approximations of the brain structure

However, the remarkable progress made during the past two decades inspired by this neurobiological analogy is really impressive

Biological and Artificial Neurons

We estimate around 10^{10} or 10^{11} the number of neurons in the human brain



Neuron

Biological and Artificial Neurons

Biological neurons (a very simplified version):

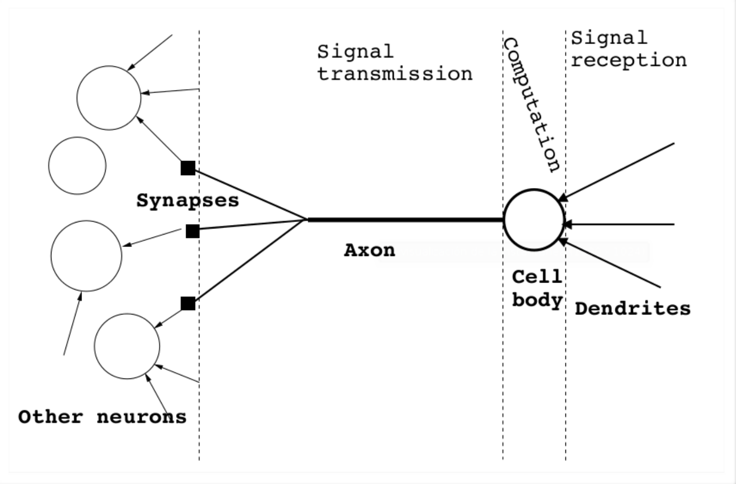
- Receive information from other neurons through the **dendrites**
- Process the information in their cell body (**soma**)
- Send information through a "cable" called an **axon**
- The point of connection between the axon branches and the dendrites of other neurons are called **synapses**
- The estimated number of synapses in the brain is 10^{15}

The "information" generated has the form of a spike in the electric potential (voltage) of the axon

Depending on the particular connections of the neurons, the information sent by one neuron can **excite** or **inhibit** other neurons

Biological and Artificial Neurons

Correspondence between artificial and biological neurons:



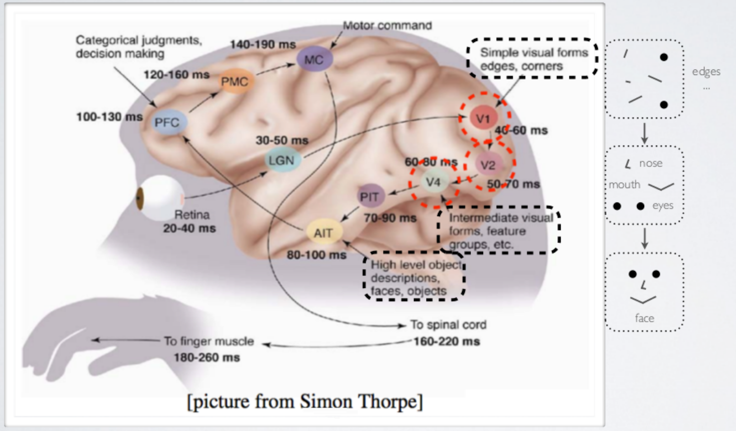
Biological and Artificial Neurons

Correspondence between artificial and biological neurons:

- The architecture models the connections among neurons: which neurons are connected to every neuron
- The weights between neurons, the aggregation and the activation functions model whether neurons excite or inhibit each other
- The output of the activation function corresponds to the information generated by the neuron

Biological Layers

The visual cortex of the brain processes the information by layers: V1, V2, V4, PIT, AIT,...



[picture from Simon Thorpe]

Many Differences between the Brain and Neural Networks

- In the brain there are many types of different neurons (pyramidal, star-shaped,...)
- The brain is specialized into many different areas (cortices), responsible for particular tasks: vision, hearing, speech, touch,... with inter-regional circuits among these areas
- Additional association areas link information affecting multiple sensations, to produce a meaningful perceptual experience of the world, enable us to interact effectively, and support abstract thinking and language
- Although biological neurons are several orders of magnitude slower than silicon logic gates (10^{-3} vs. 10^{-9} seconds), it is efficient for many tasks (by using massive interconnections between the neurons)
- In the brain, there is apparently no distinction between data and algorithms

Outline

1. Introduction
   - Definition
   - Biological Motivation
   - A Bit of History
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons

A Bit of History

1943	Logical calculus model of McCulloch & Pitts [McCulloch and Pitts, 1943]
1949	Hebb's learning for self-organizing synaptic modification [Hebb, 1949]
1958	Perceptron supervised learning model [Rosenblatt, 1958]
1960	Adaline (ADAptive LINear Element) model based on Least Mean-Squares and the delta rule [Widrow and Hoff, 1960]
1967	Minsky extends the results of McCulloch and Pitts and put them in the context of automata theory and the theory of computation [Minsky, 1967]
1969	Minsky & Papert's book *Perceptrons* [Minsky and Papert, 1969]
1970s	Self-Organizing models using competitive learning and associative memories [Anderson, 1972, Kohonen, 1972, Grossberg, 1972] and [von der Malsburg, 1973]
1980	Adaptive Resonance Theory models, started by [Grossberg, 1980]
1982	Hopfield Networks and energy-based models [Hopfield, 1982]
1985	Boltzmann Machines [Ackley et al., 1985] and Restricted Boltzmann Machines [Smolensky, 1986]

A Bit of History

1986	Backpropagation algorithm for Multilayer Perceptrons [Werbos, 1974, Rumelhart et al., 1986b] and PDP books [Rumelhart and McClelland, 1986]
1986	Auto-encoders [Rumelhart et al., 1986a]
1988	Radial-basis Function Networks [Broomhead and Lowe, 1988, Poggio and Girosi, 1990]
1988-1990	Time-Delay Neural Networks [Lang et al., 1990]
1989	Convolutional Neural Networks [LeCun et al., 1989]
1992	Support Vector Machines [Boser et al., 1992, Cortes and Vapnik, 1995]
1997	Long Short-Term Memory [Hochreiter and Schmidhuber, 1997]
1999	Contrastive Divergence [Hinton, 1999, Hinton, 2002]
2006	Deep Belief Networks [Hinton et al., 2006]
2006	Deep Neural Networks and Deep Auto-encoders [Hinton and Salakhutdinov, 2006]
2012	Large Scale Visual Recognition Challenge 2012 winner [Krizhevsky et al., 2012]

Outline

1. Introduction
2. Linear Regression
   - Normal Equations
   - Gradient Descent
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons

Linear Regression: Statement of the Problem

Type of problem: Regression

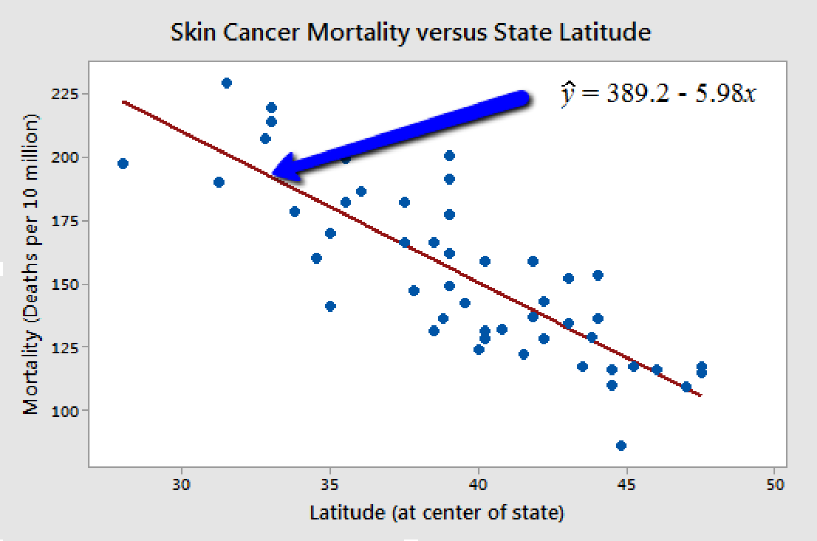
- Given a data set with N examples $\mathbf{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}$
- We want to model the relationship between the **scalar** variable y (the **target**) and the d-dimensional **vector** $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_d)$ of explanatory variables or features (the **input**)

Model representation: linear

- $h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}_0 + \boldsymbol{\theta}' \cdot \mathbf{x}$
- $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_d)'$ is the **vector of parameters**
- $\boldsymbol{\theta}_0$ is a **bias term**, that can be seen as a standard parameter if we add a constant 1 to the input $\mathbf{x} = (1, \mathbf{x}_1, \ldots, \mathbf{x}_d)$

Note that y is a scalar and $\mathbf{x}$ is a vector

Linear Regression: Statement of the Problem



Linear Regression: Statement of the Problem

Therefore, we would like that (if possible)

$$y^{(i)} = h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right) = \boldsymbol{\theta}' \cdot \mathbf{x}^{(i)} \qquad i = 1, \ldots, N$$

To that end, we define a sum-of-squares **cost function**

$$J(\boldsymbol{\theta}, \mathbf{D}) = \frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right)\right)^2 = \frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - \boldsymbol{\theta}' \cdot \mathbf{x}^{(i)}\right)^2$$

that we want to **minimize** [Legendre, 1805, Gauss, 1809]

Outline

1. Introduction
2. Linear Regression
   - Normal Equations
   - Gradient Descent
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons

Linear Regression: A First Solution (Normal Equations)

In this particular case, and given the linear nature of the problem, it can be proved that

$$J(\boldsymbol{\theta}, \mathbf{D}) = \frac{1}{2}(\mathbf{Y} - h_{\boldsymbol{\theta}}(\mathbf{X}))' \, (\mathbf{Y} - h_{\boldsymbol{\theta}}(\mathbf{X})) = \frac{1}{2}\left(\boldsymbol{\theta}' \, \mathbf{X}' \, \mathbf{X} \, \boldsymbol{\theta} - 2 \, \boldsymbol{\theta}' \, \mathbf{X}' \, \mathbf{Y} + \mathbf{Y}' \, \mathbf{Y}\right)$$

where

- $\mathbf{Y} = (y^{(1)}, \ldots, y^{(N)})'$ is the vector of target values (size $N \times 1$)
- $\mathbf{X}_{ij} = \mathbf{x}_j^{(i)}$ is the matrix of inputs (size $N \times d$)
- $h_{\boldsymbol{\theta}}(\mathbf{X}) = \mathbf{X} \; \boldsymbol{\theta}$ is the output for the whole data set

Computing the gradient of $J(\boldsymbol{\theta}, \mathbf{D})$ wrt $\boldsymbol{\theta}$ and setting it equal to 0 we have

$$\frac{\partial J(\boldsymbol{\theta}, \mathbf{D})}{\partial \boldsymbol{\theta}} = \mathbf{X}' \; \mathbf{X} \; \boldsymbol{\theta} - \mathbf{X}' \; \mathbf{Y} = 0$$

Linear Regression: A First Solution (Normal Equations)

Therefore (**normal equations**)

$$\mathbf{X}' \, \mathbf{X} \, \theta = \mathbf{X}' \, \mathbf{Y}$$

leading to

$$\theta = \left(\mathbf{X}' \, \mathbf{X}\right)^{-1} \mathbf{X}' \, \mathbf{Y}$$

The $d \times N$ matrix $\left(\mathbf{X}' \, \mathbf{X}\right)^{-1} \mathbf{X}'$ is a particular case of the the **Moore-Penrose pseudoinverse**, that generalizes the concept of inverse matrix

Even when $\left(\mathbf{X}' \, \mathbf{X}\right)^{-1}$ is singular, the normal equations can be solved, and therefore **the solution of the linear regression problem can always be computed**

Outline

1. Introduction

2. Linear Regression
   - Normal Equations
   - Gradient Descent

3. General Machine Learning Issues

4. Single-layer Perceptrons

5. Multilayer Perceptrons

Linear Regression: A Second Solution (Gradient Descent)

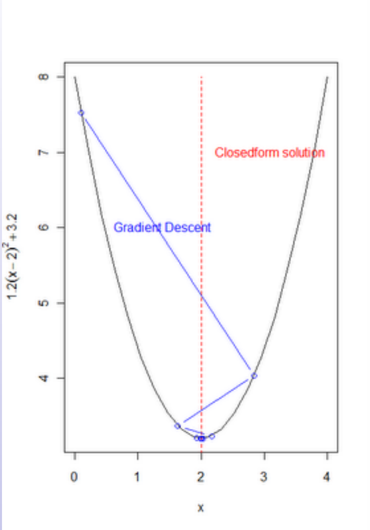
An alternative solution to the normal equations (that can be used in more general frameworks) is to perform **gradient descent in the space of the parameters**

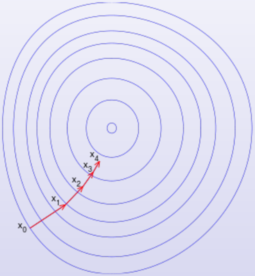
A general description of gradient descent:

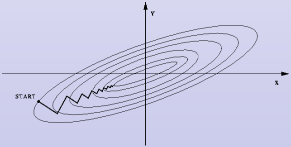
- It is a **first-order iterative optimization algorithm** for finding the minimum of a function
- It takes steps proportional to the **negative gradient** of the function (or an approximation) at the current point
- It usually obtains a **local minimum** of the function

To my knowledge, the gradient descent algorithm (also named **delta rule**) was originally described in [Hadamard, 1908], and first used in Neural Networks in [Widrow and Hoff, 1960]

Linear Regression: A Second Solution (Gradient Descent)







Linear Regression: A Second Solution (Gradient Descent)

A general algorithm for gradient descent:

```
Input: (data set $\mathbf{D}$, learning rate $\lambda > 0$)
Initialize $\boldsymbol{\theta}_0$
$t = 0$
repeat until convergence:
    Compute the gradient $g_t = \partial J(\boldsymbol{\theta}_t, \mathbf{D}) / \partial \boldsymbol{\theta}$
    $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \, g_t$
    $t = t + 1$
Return $\boldsymbol{\theta}_t$
```

Linear Regression: A Second Solution (Gradient Descent)

In the case of linear regression,

$$\partial J(\boldsymbol{\theta}_t, \mathbf{D})/\partial\boldsymbol{\theta} = \mathbf{X}' \ \mathbf{X} \ \boldsymbol{\theta}_t - \mathbf{X}' \ \mathbf{Y} = \mathbf{X}' \left(h_{\boldsymbol{\theta}_t}(\mathbf{X}) - \mathbf{Y} \right)$$

Since the linear regression problem leads to a convex surface, **the gradient descent algorithm converges to its global minimum** [Horst and Tuy, 1993]

Linear Regression: Comparison of Solutions

Advantages of the normal equations:

- The solution is obtained in one step, without iterations
- There are no parameters to select (learning rate)

Advantages of the gradient descent algorithm:

- It does not present numerical problems related to the inverse of $\left(\mathbf{X}'\,\mathbf{X}\right)^{-1}$
- It is faster when the number of features d is large, since the computation of $\left(\mathbf{X}'\,\mathbf{X}\right)^{-1}$ is $O(d^3)$
- **It is a general framework that can be used in other problems**

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
   - Type of Problem, Model Representation, Cost Function and Optimization Technique
   - Examples
4. Single-layer Perceptrons
5. Multilayer Perceptrons

Outline

1. Introduction

2. Linear Regression

3. General Machine Learning Issues
   - Type of Problem, Model Representation, Cost Function and Optimization Technique
   - Examples

4. Single-layer Perceptrons

5. Multilayer Perceptrons

Type of Problem, Model Representation, Cost Function and Optimization Technique

The previously explained example of linear regression contains all the basic ingredients needed in a problem that is going to be tackled with Neural Networks:

- Type of Problem
- Model Representation
- Cost Function
- Optimization Technique

Not specific for Neural Networks, but for Machine Learning models in general

Type of Problem, Model Representation, Cost Function and Optimization Technique

Suppose that we are given a data set with N examples $\mathbf{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}$ from which we want to learn something (this is the aim of **Machine Learning**)

The **Type of Problem** is the nature of the problem we are trying to solve: regression, classification, clustering, etc

- In the case of linear regression, it is regression

The **Model Representation** is the form of the output function (hypothesis) of the model

- In the case of linear regression, it is a linear function that depends on several parameters $h_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \boldsymbol{\theta}' \cdot \mathbf{x}$

Type of Problem, Model Representation, Cost Function and Optimization Technique

The **Cost Function** is the (usually scalar) function to be optimized, which **strongly depends on the type of problem** (typically based on the **empirical risk minimization**) and is a **function of the data and the parameters**

- In the case of linear regression, it is the sum-of-squares error function $J(\theta, \mathbf{D}) = \frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right)\right)^2$

The **Optimization Technique** is used to optimize the cost function

- In the case of linear regression, it can be done with the normal equations, gradient descent, etc

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
    - Type of Problem, Model Representation, Cost Function and Optimization Technique
    - Examples
4. Single-layer Perceptrons
5. Multilayer Perceptrons

Ridge Regression

Ridge Regression is a regularized version of linear regression, that can be represented as:

- **Type of problem**: regression
- **Model representation**: the same as linear regression

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}_0 + \boldsymbol{\theta}' \cdot \mathbf{x}$$

- **Cost function**: (2-norm) regularized sum-of-squares error

$$J(\boldsymbol{\theta}, \mathbf{D}) = \frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right)\right)^2 + \alpha\sum_{j=0}^{d}\boldsymbol{\theta}_j^2$$

- **Optimization technique**: modified normal equations, gradient descent

Ridge Regression

By the way, it can be proved that minimizing the cost function of ridge regression

$$J(\boldsymbol{\theta}, \mathbf{D}) = \frac{1}{2} \sum_{i=1}^{N} \left(y^{(i)} - h_{\boldsymbol{\theta}} \left(\mathbf{x}^{(i)} \right) \right)^2 + \alpha \sum_{j=0}^{d} \boldsymbol{\theta}_j^2$$

is equivalent to solving the following constrained optimization problem:

$$\begin{array}{ll} \text{Minimize}_{\boldsymbol{\theta}} & \frac{1}{2} \sum_{i=1}^{N} \left(y^{(i)} - h_{\boldsymbol{\theta}} \left(\mathbf{x}^{(i)} \right) \right)^2 \\ \text{subject to} & \sum_{j=0}^{d} \boldsymbol{\theta}_j^2 \leqslant T \end{array}$$

which is a form of **structural risk minimization**

Lasso Regression

Lasso Regression is another regularized version of linear regression, that can be represented as:

- **Type of problem**: regression
- **Model representation**: the same as linear regression

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}_0 + \boldsymbol{\theta}' \cdot \mathbf{x}$$

- **Cost function**: (1-norm) regularized sum-of-squares error

$$J(\boldsymbol{\theta}, \mathbf{D}) = \frac{1}{2} \sum_{i=1}^{N} \left(y^{(i)} - h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right) \right)^2 + \alpha \sum_{j=0}^{d} |\boldsymbol{\theta}_j|$$

- **Optimization technique**: quadratic programming techniques, least angle regression for Lasso

Lasso Regression

Similar to ridge regression, it can be proved that minimizing the cost function of lasso regression

$$J(\boldsymbol{\theta}, \mathbf{D}) = \frac{1}{2} \sum_{i=1}^{N} \left(y^{(i)} - h_{\boldsymbol{\theta}} \left(\mathbf{x}^{(i)} \right) \right)^2 + \alpha \sum_{j=0}^{d} |\boldsymbol{\theta}_j|$$

is equivalent to solving the following constrained optimization problem:

$$\begin{array}{ll} \text{Minimize}_{\boldsymbol{\theta}} & \frac{1}{2} \sum_{i=1}^{N} \left(y^{(i)} - h_{\boldsymbol{\theta}} \left(\mathbf{x}^{(i)} \right) \right)^2 \\ \text{subject to} & \sum_{j=0}^{d} |\boldsymbol{\theta}_j| \leqslant T \end{array}$$

Support Vector Machines for Classification

In (1-norm soft margin) Support Vector Machines for classification we have [Cristianini and Shawe-Taylor, 2000]

- **Type of problem**: classification
- **Model representation**: $f_{SVM}(\mathbf{x}) = b + \sum_{i=1}^{N} \alpha_i y^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x})$, where $\boldsymbol{\theta} = (b, \{\alpha_i\}_{i=1}^{N})$
- **Cost function**:

$$\begin{array}{lll} \text{Maximize}_{\alpha} & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y^{(i)} y^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) & \\ \text{subject to} & \sum_{i=1}^{N} \alpha_i y^{(i)} = 0 & \text{(bias constraint)} \\ & 0 \leqslant \alpha_i \leqslant C & i = 1 \dots N \end{array}$$

- **Optimization technique**: coordinate descent, gradient projection, decomposition variants (SMO, SMV,...), interior point, primal-dual approaches, etc

Bayesian Networks

In Bayesian Networks and other Probabilistic Graphical Models we have [Koller and Friedman, 2009]

- **Type of problem**: probability estimation
- **Model representation**: joint probability function $P\left(\mathbf{x};\boldsymbol{\theta}\right)$, with different constraints
- **Cost function**: log-likelihood of the data (maximize):

$$J(\boldsymbol{\theta},\mathbf{D}) = \sum_{i=1}^{N} \log P\left(\mathbf{x}^{(i)};\boldsymbol{\theta}\right)$$

- **Optimization technique**: gradient ascent, expectation-maximization, etc

Regression Trees

In the standard formulation of Regression Trees we have [Hastie et al., 2001]

- **Type of problem**: regression
- **Model representation**: binary trees
- **Cost function**: sum-of-squares error (minimize)
- **Optimization technique**: since finding the best partition in terms of the quadratic error is computationally unfeasible, greedy algorithms are performed, for example selecting a splitting variable j and a splitting point s such that minimize

$$\min_{j,s}\left(\min_{c_1}\sum_{\mathbf{x}^{(i)}\in R_1(j,s)}(y^{(i)}-c_1)^2+\min_{c_2}\sum_{\mathbf{x}^{(i)}\in R_2(j,s)}(y^{(i)}-c_2)^2\right)$$

where $R_1(j,s)=\{\mathbf{x}\mid \mathbf{x}_j\leqslant s\}$ and $R_2(j,s)=\{\mathbf{x}\mid \mathbf{x}_j> s\}$

Neural Networks

Neural Networks can also be included in this framework:

- **Type of problem**: classification, regression, etc
- **Model representation**: the particular architecture for the problem at hand: Single-layer Perceptron, Multilayer Perceptron, Radial-basis Function Network, Convolutional Neural Network, Recurrent Neural Network, etc
- **Cost function**: depends on the type of problem (sum-of-squares error, cross-entropy, etc) with or without regularization
- **Optimization technique**: gradient descent (with many variations), etc

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
   - Definition
   - Rosenblatt's Perceptron
   - Linear Regression
   - Logistic Regression
5. Multilayer Perceptrons

Outline

1 Introduction

2 Linear Regression

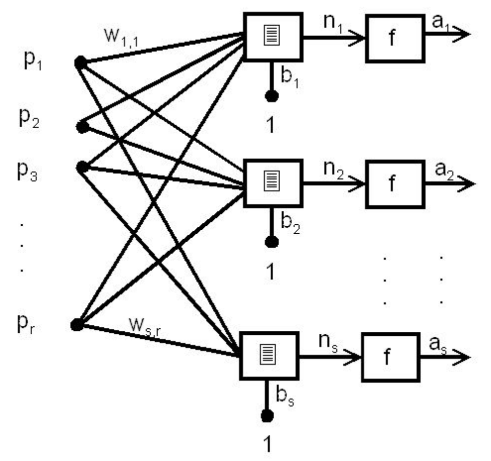
3 General Machine Learning Issues

4 Single-layer Perceptrons
- Definition
- Rosenblatt's Perceptron
- Linear Regression
- Logistic Regression

5 Multilayer Perceptrons

Definition

Single-layer perceptrons are neural networks which only have one layer of weights



Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
   - Definition
   - Rosenblatt's Perceptron
   - Linear Regression
   - Logistic Regression
5. Multilayer Perceptrons

Rosenblatt's Perceptron

M. Rosenblatt was a psychologist (Neural Networks were born in these fields: W. S. McCulloch was a neurophysiologist, W. H. Pitts was a logician and D. O. Hebb was also a psychologist)

To my knowledge, Rosenblatt's Perceptron [Rosenblatt, 1958] is the first algorithmically described Neural Network, and the first that was implemented in hardware (Mark I Perceptron, 1960)

It is a supervised learning linear model consisting of a single neuron with adjustable synaptic weights and bias

It has several similarities with Support Vector Machines

Rosenblatt's Perceptron

The Perceptron algorithm [Rosenblatt, 1958]:

```
Input: (data $\mathbf{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}$), $y^{(i)} \in \{-1, +1\}$
$\omega = 0$ ($\omega$ includes the bias term, with an additional 1 in $\mathbf{x}^{(i)}$)
repeat
    for i from 1 to N (for every example):
        Compute the output of $\mathbf{x}^{(i)}$:   $h(\mathbf{x}^{(i)}) = \text{sgn}\left(\omega \cdot \mathbf{x}^{(i)}\right)$
        if $y^{(i)} \cdot h(\mathbf{x}^{(i)}) < 0$   // example bad classified
            $\omega = \omega + \eta \cdot y^{(i)} \cdot \mathbf{x}^{(i)}$
until no mistakes are made in the for loop
Return $\omega$
```

Rosenblatt's Perceptron

Therefore we have:

- **Type of problem**: Classification

  We want to assign to a d-dimensional **input vector** $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_d)$ its corresponding **class** y
- **Model representation**: $h_\omega(\mathbf{x}) = \operatorname{sgn}\left(\omega \cdot \mathbf{x}^{(i)}\right)$
- **Cost function**: the number of mistakes (classification accuracy)
- **Optimization technique**: the algorithm previously described

Rosenblatt's Perceptron

It can be proved that if the patterns (vectors) used to train are linearly separable, then the algorithm converges and its decision surface (an hyperplane) separates the two classes

It could be seen as an algorithm of "pseudo gradient descent" with the objective of minimizing a (non-differenciable) function of the functional margin

$$\sum_{i=1}^{N} \max\left(0, -y^{(i)} \cdot \omega \cdot \mathbf{x}^{(i)}\right)$$

Outline

1. Introduction

2. Linear Regression

3. General Machine Learning Issues

4. Single-layer Perceptrons
   - Definition
   - Rosenblatt's Perceptron
   - Linear Regression
   - Logistic Regression

5. Multilayer Perceptrons

Linear Regression

Although it is not the most common interpretation, the standard version of linear regression (trained with gradient descent), can be seen as a neural networks problem:

- **Type of problem**: regression
- **Model representation**: $h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}_0 + \boldsymbol{\theta}' \cdot \mathbf{x}$
  (a single-layer MLP with linear output units)
- **Cost function**: sum-of-squares error

$$J(\boldsymbol{\theta}, \mathbf{D}) = \frac{1}{2} \sum_{i=1}^{N} \left(y^{(i)} - h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right) \right)^2$$

- **Optimization technique**: normal equations, gradient descent with

$$\partial J(\boldsymbol{\theta}_t, \mathbf{D}) / \partial \boldsymbol{\theta} = \mathbf{X}'\ \mathbf{X}\ \boldsymbol{\theta}_t - \mathbf{X}'\ \mathbf{Y} = \mathbf{X}' \left(h_{\boldsymbol{\theta}_t}(\mathbf{X}) - \mathbf{Y} \right)$$

Outline

1 Introduction

2 Linear Regression

3 General Machine Learning Issues

4 Single-layer Perceptrons
- Definition
- Rosenblatt's Perceptron
- Linear Regression
- Logistic Regression

5 Multilayer Perceptrons

Logistic Regression

Type of problem: Classification

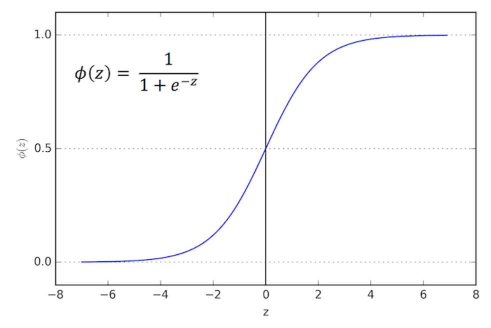
In order to model it with neural networks, we can assume that (or transform to) $y^{(i)} \in \{0,1\} \quad i = 1,\ldots,N$

Note that, despite its name, **logistic regression refers to a classification problem**

Logistic Regression

Model representation: a single-layer MLP with one logistic output unit

- $h_{\boldsymbol{\theta}}(\mathbf{x}) = f\left(\boldsymbol{\theta}' \cdot \mathbf{x}\right)$
- $f(z) = \frac{1}{1+e^{-z}}$ is the logistic function



Logistic Regression

Interpretation: The output $h_{\boldsymbol{\theta}}(\mathbf{x})$ is the conditional probability that $\mathbf{x}$ belongs to class 1: $h_{\boldsymbol{\theta}}(\mathbf{x}) = P(y = 1 \mid \mathbf{x}; \boldsymbol{\theta})$

(linear output units could give values not in $[0, 1]$)

Note that

- The **decision boundary** is linear: $\boldsymbol{\theta}' \cdot \mathbf{x} = 0$
- $\log\left(\frac{P(y=1 \mid \mathbf{x};\boldsymbol{\theta})}{P(y=0 \mid \mathbf{x};\boldsymbol{\theta})}\right) = \boldsymbol{\theta}' \cdot \mathbf{x}$

Logistic Regression

Cost function: Cross-entropy

$$J(\boldsymbol{\theta}, \mathbf{D}) = \sum_{i=1}^{N} -y^{(i)} \cdot \log\left(h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right)\right) + \\ + \sum_{i=1}^{N} -(1 - y^{(i)}) \cdot \log\left(1 - h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right)\right)$$

equivalent to **negative log-likelihood** (see below, Training MLPs)

The classification accuracy is not a smooth function!

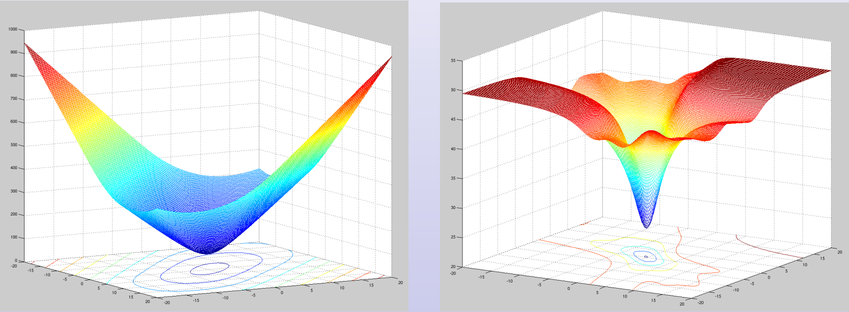
Interpretation:

- When $y^{(i)} = 0$, we only have $-\log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))$, that will be small/large if $h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})$ is near 0/1
- When $y^{(i)} = 1$, we only have $-\log(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))$, that will be small/large if $h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})$ is near 1/0

Logistic Regression

Why not using the sum-of-squares error function in this case?

Because **the cross-entropy is convex wrt the parameters**, whereas the sum-of-squares error function is not



Left/Right: cross-entropy/sum-of-squares error function

Logistic Regression

Optimization technique: gradient descent

The gradient of the cross-entropy error function wrt the parameters **has the same form than the gradient of the sum-of-squares error**

$$\partial J(\boldsymbol{\theta}_t, \mathbf{D}) / \partial \boldsymbol{\theta} = \mathbf{X}' \left(h_{\boldsymbol{\theta}_t}(\mathbf{X}) - \mathbf{Y} \right)$$

(you can check it!)

Outline

1. Introduction

2. Linear Regression

3. General Machine Learning Issues

4. Single-layer Perceptrons

5. Multilayer Perceptrons
    - Definition
    - Training MLPs
    - The Backpropagation Algorithm
    - Gradient Descent with Backpropagation
    - Theoretical Results

Outline

1. Introduction

2. Linear Regression

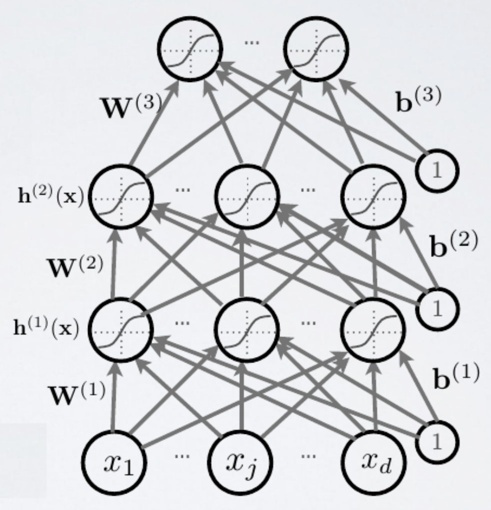
3. General Machine Learning Issues

4. Single-layer Perceptrons

5. Multilayer Perceptrons
    - Definition
    - Training MLPs
    - The Backpropagation Algorithm
    - Gradient Descent with Backpropagation
    - Theoretical Results

Definition

An example of a MLP with 3 layers of weights



Definition

Multilayer perceptrons (MLPs) are neural networks with several layers of weights, characterized by the fact that the aggregation function is the inner product between the weights and the activation of the previous units

As usual, we will consider the bias terms as standard weights by adding a constant unit of value 1 to the input of every layer

The computation starts in the input layer, and proceeds **layer by layer**: in order to compute the activations of the units a certain layer, **we only need to know the activations of the units of the previous layer** (and the weights of that layer, obviously)

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons
    - Definition
    - Training MLPs
    - The Backpropagation Algorithm
    - Gradient Descent with Backpropagation
    - Theoretical Results

Training MLPs

For MLPs the most common settings are:

- **Type of problem**: regression, classification
- **Model representation**: the typical architecture of an MLP, where the units are structured by layers
- **Cost function**:
  - Sum-of-squares error for regression problems
  - Cross-entropy for classification problems
  - With or without regularization terms
- **Optimization technique**: based on gradient descent with backpropagation

A Note on Multiclass Classification Problems

For multiclass classification problems

- The activation function of the output units usually is the softmax function $f(z)_c = \frac{e^{z_c}}{\sum_k e^{z_k}}$ (as many outputs as classes)
- We interpret every output c of the network as the probability of belonging to that particular class:

$$h_{\boldsymbol{\theta}}(\mathbf{x})_c = P(y = c \mid \mathbf{x}; \boldsymbol{\theta})$$

In this setting, the cross-entropy is defined as

$$J(\boldsymbol{\theta}, \mathbf{D}) = \sum_{i=1}^{N} \sum_{c} -1_{(y^{(i)}=c)} \cdot \log h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}\right)_c$$

(we take the logarithm for numerical stability and math simplicity) which is equivalent to the **negative log-likelihood**

$$\sum_{i=1}^{N} -\log P(y = y^{(i)} \mid \mathbf{x}^{(i)}; \boldsymbol{\theta})$$

and related to the **Kullback-Leibler divergence**

A Note on Multiclass Classification Problems

Instead of $-1_{(y^{(i)}=c)}$, we use a 1-of-C coding scheme for the target variable: a vector of C components $(y_1^{(i)}, \dots, y_C^{(i)})$ where the position of the correct class is 1 and all others are 0

In this setting, the cross-entropy can be redefined as

$$J(\boldsymbol{\theta}, \mathbf{D}) = \sum_{i=1}^{N} \sum_{c=1}^{C} -y_c^{(i)} \cdot \log h_{\boldsymbol{\theta}} \left(\mathbf{x}^{(i)} \right)_c$$

(This will be useful for the analytic computation of the gradient)

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons
   - Definition
   - Training MLPs
   - The Backpropagation Algorithm
     - Derivation of the Backpropagation Algorithm: An Example
     - Derivation of the Backpropagation Algorithm: Efficient Algorithm
   - Gradient Descent with Backpropagation
   - Theoretical Results

Remember Gradient Descent

A general algorithm for gradient descent:

```
Input: (data set $\mathbf{D}$, learning rate $\lambda > 0$)
Initialize $\boldsymbol{\theta}_0$
$t = 0$
**repeat until convergence**:
    Compute the gradient $g_t = \partial J(\boldsymbol{\theta}_t, \mathbf{D}) / \partial \boldsymbol{\theta}$
    $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \, g_t$
    $t = t + 1$
Return $\boldsymbol{\theta}_t$
```

Definition

Recall that in order to apply gradient descent we only need:

- A way to initialize $\boldsymbol{\theta}_0$ $(t = 0)$
- A way to compute the gradient of the cost function $g_t = \partial J(\boldsymbol{\theta}_t, \mathbf{D})/\partial\boldsymbol{\theta}$

Backpropagation (BP) is an **efficient algorithm** for the computation of the derivative of the cost function of an MLP with respect to its weights

It is not an optimization technique: backpropagation can be used with **ANY** optimization technique that needs the derivative of the most usual cost functions

Definition

BP is a **clever application of the chain rule**, which states that the derivative of the composition of two functions is expressed as

$$\frac{\partial(f_\theta \circ g_\theta)}{\partial \boldsymbol{\theta}}(\mathbf{x}) = \frac{\partial f_\theta}{\partial \boldsymbol{\theta}}\left(g_\theta\left(\mathbf{x}\right)\right) \cdot \frac{\partial g_\theta}{\partial \boldsymbol{\theta}}(\mathbf{x})$$

It profits the fact that, in order to compute the activation of a unit, we only need the information of the units in the previous layer, which allows to **think recursively** the problem: compute the derivatives of the weights in the output layer, and then **back-propagate** them (layer by layer) to the previous layers

It can be extended to other architectures just by taking into account the information of the units connected to a certain unit

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons
    - Definition
    - Training MLPs
    - The Backpropagation Algorithm
        - Derivation of the Backpropagation Algorithm: An Example
        - Derivation of the Backpropagation Algorithm: Efficient Algorithm
    - Gradient Descent with Backpropagation
    - Theoretical Results

Derivation of the Backpropagation Algorithm: Example

A bit of notation:

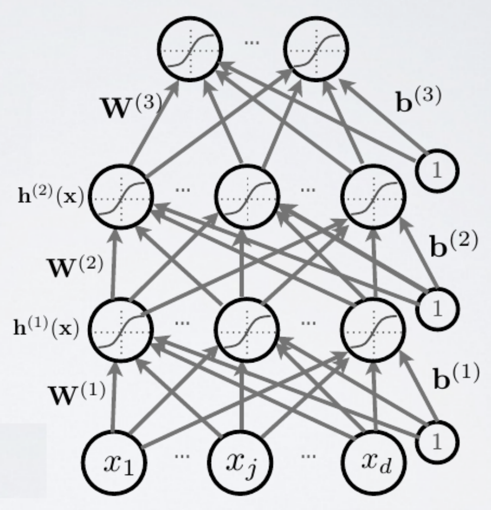
- The layers of the network will be enumerated as $0, \cdots, L$, where 0 is the input layer, and L is the output layer
- $\mathbf{W}^{(l)}$ will denote the weights matrix in layer l ($l \geqslant 1$)
- $\mathbf{W}_{ij}^{(l)}$ will denote the weight connecting unit i of layer l with unit j of layer $l - 1$
- $n_i^{(l)}$: Pre-activation (net input) of the unit i in layer l
- $g_i^{(l)}$: Activation function of the unit i in layer l
- $h_i^{(l)}$: Activation (output) of the unit i in layer l

With this notation we have

- $n_i^{(l)} = \sum_j \mathbf{W}_{ij}^{(l)} h_j^{(l-1)} \qquad l \geqslant 1$
- $h_i^{(l)} = g_i^{(l)}(n_j^{(l)}) \qquad h_i^{(0)} = \mathbf{x}_i$ for the input layer ($l = 0$)

Derivation of the Backpropagation Algorithm: Example

Consider the example of a MLP with 3 layers of weights



Derivation of the Backpropagation Algorithm: Example

In this example, we will use

- Index t for units in layer 0 (input layer)
- Index s for units in layer 1
- Index r for units in layer 2
- Index c for units in layer 3 (output layer)

Therefore,

- Each input example $\mathbf{x}$ will feed the activations of the units in the input layer: $h_t^{(0)} = \mathbf{x}_t$
- Information will be propagated through the layers in order to obtain the pre-activations $n_c^{(3)}$ of the output units in layer 3 (in general, we will have a vector of outputs)

Derivation of the Backpropagation Algorithm: Example

In detail, the forward computation of an input example $\mathbf{x}$ would be

- For $l = 0$, $h_t^{(0)} = \mathbf{x}_t$
- For $l = 1$, $n_s^{(1)} = \sum_t \mathbf{W}_{st}^{(1)} h_t^{(0)} = \sum_t \mathbf{W}_{st}^{(1)} \mathbf{x}_t$
- For $l = 2$, $n_r^{(2)} = \sum_s \mathbf{W}_{rs}^{(2)} h_s^{(1)} = \sum_s \mathbf{W}_{rs}^{(2)} g_s^{(1)}(n_s^{(1)})$
- For $l = 3$, $n_c^{(3)} = \sum_r \mathbf{W}_{cr}^{(3)} h_r^{(2)} = \sum_r \mathbf{W}_{cr}^{(3)} g_r^{(2)}(n_r^{(2)})$

Therefore, the whole computation of the pre-activations of the output units can be expressed as a composition of functions

$$n_c^{(3)} = \sum_r \mathbf{W}_{cr}^{(3)} g_r^{(2)} \left(\sum_s \mathbf{W}_{rs}^{(2)} g_s^{(1)} \left(\sum_t \mathbf{W}_{st}^{(1)} \mathbf{x}_t \right) \right) \qquad (1)$$

Derivation of the Backpropagation Algorithm: Example

The output function of every output unit of the network will be

- For regression problems, $h_c^{(3)} = n_c^{(3)}$
- For classification problems,

$$h_c^{(3)} = \mathrm{softmax}(n_{\cdot}^{(3)})_c = \frac{e^{n_c^{(3)}}}{\sum_{k=1}^{C} e^{n_k^{(3)}}}$$

Note that $h_c^{(3)}$ is a function of $\mathbf{x}$ and the weights of the network:

$$h_c^{(3)} = h_c^{(3)}(\mathbf{W}, \mathbf{x}) = h_{\boldsymbol{\theta}}(\mathbf{x})_c$$

(the last equality is the notation consistent with the rest of the slides)

Derivation of the Backpropagation Algorithm: Example

The cost function $J(\boldsymbol{\theta}, \mathbf{D})$ will be computed with $h_c^{(3)} = h_{\boldsymbol{\theta}}(\mathbf{x})_c$

- The sum-of-squares error for regression problems

$$J(\boldsymbol{\theta}, \mathbf{D}) = \sum_{i=1}^{N} \sum_{c=1}^{C} \frac{1}{2} \left(y_c^{(i)} - h_{\boldsymbol{\theta}} \left(\mathbf{x}^{(i)} \right)_c \right)^2 \qquad (2)$$

  (C is the number of targets/outputs)

- The cross-entropy for classification

$$J(\boldsymbol{\theta}, \mathbf{D}) = \sum_{i=1}^{N} \sum_{c=1}^{C} -y_c^{(i)} \cdot \log h_{\boldsymbol{\theta}} \left(\mathbf{x}^{(i)} \right)_c \qquad (3)$$

  with a 1-of-C coding scheme for the target variable (C is the number of classes/targets/outputs)

Derivation of the Backpropagation Algorithm: Example

We can compute the derivative of the cost function wrt a particular weights by applying the chain rule on equation (1)

For example, for regression problems (we omit the sum over the data set **D** for the sake of clarity),

$$\frac{\partial J(\boldsymbol{\theta},\mathbf{x})}{\partial \mathbf{W}^{(2)}_{r_1 s_1}} = \frac{\partial}{\partial \mathbf{W}^{(2)}_{r_1 s_1}} \sum_{c=1}^{C} \frac{1}{2}\left(y_c - h_{\boldsymbol{\theta}}\left(\mathbf{x}\right)_c\right)^2 = \sum_{c=1}^{C}\left(y_c - n^{(3)}_c(\mathbf{x})\right)\frac{\partial n^{(3)}_c(\mathbf{x})}{\partial \mathbf{W}^{(2)}_{r_1 s_1}}$$

with

$$\begin{aligned}\frac{\partial n^{(3)}_c(\mathbf{x})}{\partial \mathbf{W}^{(2)}_{r_1 s_1}} &= \frac{\partial}{\partial \mathbf{W}^{(2)}_{r_1 s_1}} \sum_r \mathbf{W}^{(3)}_{cr}\, g^{(2)}_r\left(\sum_s \mathbf{W}^{(2)}_{rs}\, g^{(1)}_s\left(\sum_t \mathbf{W}^{(1)}_{st}\mathbf{x}_t\right)\right) \\ &= \mathbf{W}^{(3)}_{cr_1}\, g^{(2)\prime}_{r_1}\left(\sum_s \mathbf{W}^{(2)}_{r_1 s}\, g^{(1)}_s\left(\sum_t \mathbf{W}^{(1)}_{st}\mathbf{x}_t\right)\right) g^{(1)}_{s_1}\left(\sum_t \mathbf{W}^{(1)}_{s_1 t}\mathbf{x}_t\right)\end{aligned}$$

Derivation of the Backpropagation Algorithm: Example

Now we compute the derivative of the cost function wrt **every weights** in **every layer** applying the chain rule in a more elegant way:

$$\frac{\partial J}{\partial \mathbf{W}_{cr}^{(3)}} = \frac{\partial J}{\partial n_c^{(3)}} \frac{\partial n_c^{(3)}}{\partial \mathbf{W}_{cr}^{(3)}}$$

$$\frac{\partial J}{\partial \mathbf{W}_{rs}^{(2)}} = \frac{\partial J}{\partial n_r^{(2)}} \frac{\partial n_r^{(2)}}{\partial \mathbf{W}_{rs}^{(2)}}$$

$$\frac{\partial J}{\partial \mathbf{W}_{st}^{(1)}} = \frac{\partial J}{\partial n_s^{(1)}} \frac{\partial n_s^{(1)}}{\partial \mathbf{W}_{st}^{(1)}}$$

Note that **we apply the chain rule to "the first element after the weight"** (the first time the weight has any influence in the network)

Derivation of the Backpropagation Algorithm: Example

The second element of every derivative is very easy to compute. Thus:

$$\frac{\partial J}{\partial \mathbf{W}^{(3)}_{cr}} = \frac{\partial J}{\partial n^{(3)}_c} h^{(2)}_r$$
$$\frac{\partial J}{\partial \mathbf{W}^{(2)}_{rs}} = \frac{\partial J}{\partial n^{(2)}_r} h^{(1)}_s$$
$$\frac{\partial J}{\partial \mathbf{W}^{(1)}_{st}} = \frac{\partial J}{\partial n^{(1)}_s} h^{(0)}_t$$

Note that $h^{(0)}_t = \mathbf{x}_t$

Derivation of the Backpropagation Algorithm: Example

Now we apply again the chain rule to every first term:

$$\frac{\partial J}{\partial \mathbf{W}_{cr}^{(3)}} = \frac{\partial J}{\partial n_c^{(3)}} h_r^{(2)}$$

$$\frac{\partial J}{\partial \mathbf{W}_{rs}^{(2)}} = \frac{\partial J}{\partial n_r^{(2)}} h_s^{(1)} = \sum_c \frac{\partial J}{\partial n_c^{(3)}} \frac{\partial n_c^{(3)}}{\partial n_r^{(2)}} h_s^{(1)}$$

$$\frac{\partial J}{\partial \mathbf{W}_{st}^{(1)}} = \frac{\partial J}{\partial n_s^{(1)}} h_t^{(0)} = \sum_c \frac{\partial J}{\partial n_c^{(3)}} \frac{\partial n_c^{(3)}}{\partial n_s^{(1)}} h_t^{(0)} = \sum_c \frac{\partial J}{\partial n_c^{(3)}} \sum_r \frac{\partial n_c^{(3)}}{\partial n_r^{(2)}} \frac{\partial n_r^{(2)}}{\partial n_s^{(1)}} h_t^{(0)}$$

Why? Remember that

- $J = \frac{1}{2} \sum_{c=1}^{C} \left(y_c - n_c^{(3)}(\mathbf{x}) \right)^2$
- $n_c^{(3)} = \sum_r \mathbf{W}_{cr}^{(3)} g_r^{(2)}(n_r^{(2)})$
- $n_r^{(2)} = \sum_s \mathbf{W}_{rs}^{(2)} g_s^{(1)}(n_s^{(1)})$

Derivation of the Backpropagation Algorithm: Example

The partial derivatives between pre-activations are also very easy to compute:

$$\frac{\partial n_c^{(3)}}{\partial n_r^{(2)}} = \mathbf{W}_{cr}^{(3)} {g'}_r^{(2)}(n_r^{(2)})$$

$$\frac{\partial n_r^{(2)}}{\partial n_s^{(1)}} = \mathbf{W}_{rs}^{(2)} {g'}_s^{(1)}(n_s^{(1)})$$

Therefore, the only thing we have not computed yet is the derivative of the cost function in the last layer: $\frac{\partial J}{\partial n_c^{(3)}}$, which depends on the particular cost function (see below, Efficient Algorithm)

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons
    - Definition
    - Training MLPs
    - The Backpropagation Algorithm
        - Derivation of the Backpropagation Algorithm: An Example
        - Derivation of the Backpropagation Algorithm: Efficient Algorithm
    - Gradient Descent with Backpropagation
    - Theoretical Results

Derivation of the Backpropagation Algorithm: Efficient

Alternatively, **we can proceed in a dynamic way in order to make the algorithm more efficient**, by using the computations made in a layer for the previous one (as in the forward step, but in the opposite direction)

Let us define (after rearranging terms) the **local gradients** as

$$\delta_c^{(3)} = \frac{\partial J}{\partial n_c^{(3)}}$$

$$\delta_r^{(2)} = \frac{\partial J}{\partial n_r^{(2)}} = \sum_c \frac{\partial J}{\partial n_c^{(3)}} \frac{\partial n_c^{(3)}}{\partial n_r^{(2)}} = \cdots = {g'}_r^{(2)}(n_r^{(2)}) \sum_c \delta_c^{(3)} \mathbf{W}_{cr}^{(3)}$$

$$\delta_s^{(1)} = \frac{\partial J}{\partial n_s^{(1)}} = \sum_c \frac{\partial J}{\partial n_c^{(3)}} \sum_r \frac{\partial n_c^{(3)}}{\partial n_r^{(2)}} \frac{\partial n_r^{(2)}}{\partial n_s^{(1)}} = \cdots = {g'}_s^{(1)}(n_s^{(1)}) \sum_r \delta_r^{(2)} \mathbf{W}_{rs}^{(2)}$$

which can be computed layer-wise (dynamically)

Derivation of the Backpropagation Algorithm: Efficient

With the previous definitions we have a very easy expression for the derivative of the cost function

$$\frac{\partial J}{\partial \mathbf{W}_{cr}^{(3)}} = \delta_c^{(3)} h_r^{(2)}$$

$$\frac{\partial J}{\partial \mathbf{W}_{rs}^{(2)}} = \delta_r^{(2)} h_s^{(1)}$$

$$\frac{\partial J}{\partial \mathbf{W}_{st}^{(1)}} = \delta_s^{(1)} h_t^{(0)}$$

that can be computed **LAYER BY LAYER** starting from $\delta_c^{(3)} = \frac{\partial J}{\partial n_c^{(3)}}$

We assume that we have stored in memory the values of the pre-activations and activations of the units in the network

Derivation of the Backpropagation Algorithm: Efficient

What is the expression for $\delta_c^{(3)} = \frac{\partial J}{\partial n_c^{(3)}}$?

It depends on the particular cost function, but for the most usual ones it is also very easy:

- For regression problems with the sum-of-squares error (2) and linear output units

$$\delta_c^{(3)} = h_c^{(3)} - y_c$$

- For regression problems with the sum-of-squares error (2) and output units with activation function $g_c^{(3)}$

$$\delta_c^{(3)} = (h_c^{(3)} - y_c) \cdot {g'}_c^{(3)}(n_c^{(3)})$$

- For classification problems with the cross-entropy cost function (3) and softmax output units

$$\delta_c^{(3)} = h_c^{(3)} - y_c$$

(you can check it!)

Outline

1. Introduction
2. Linear Regression
3. General Machine Learning Issues
4. Single-layer Perceptrons
5. Multilayer Perceptrons
    - Definition
    - Training MLPs
    - The Backpropagation Algorithm
    - Gradient Descent with Backpropagation
    - Theoretical Results

Gradient Descent with Backpropagation

A general algorithm for the computation of the gradient with BP:

```
Input: (data $\mathbf{x}$)
Perform a forward pass with $\mathbf{x}$ to compute $n_\cdot^{(\cdot)}$ and $h_\cdot^{(\cdot)}$
Compute the local gradients of the last layer $\delta_\cdot^{(L)} = \frac{\partial J}{\partial n_\cdot^{(L)}}$
Compute the gradients of the last layer
  $\partial J / \partial \mathbf{W}_{ij}^{(L)} = \delta_i^{(L)} h_j^{(L-1)}$
for l from L-1 to 1:
  Compute the local gradient of the current layer
    $\delta_m^{(l)} = g'^{(l)}_m (n_m^{(l)}) \sum_k \delta_k^{(l+1)} \mathbf{W}_{km}^{(l+1)}$
  Compute the gradients of the current layer
    $\partial J / \partial \mathbf{W}_{ij}^{(l)} = \delta_i^{(l)} h_j^{(l-1)}$
Return $\partial J / \partial \mathbf{W}_\cdot^{(\cdot)}$
```

Gradient Descent with Backpropagation

Note that the BP algorithm is an example of **Dynamic Programming** [Bellman, 1957]

The BP algorithm can be easily implemented with products of matrices, allowing **efficient vectorized or GPU implementations** (theano, tensorflow, julia, matlab, etc)

Exercise: Describe a vectorized version (no loops except the main one) of BP for an input matrix $\mathbf{X}$ ($\mathbf{X}_{ij} = \mathbf{x}_j^{(i)}$) with size $N \times d$

Although MLPs can be trained with other schemes different from gradient descent (Conjugate Gradients, Levenberg-Marquardt, etc), it is most likely that these methods need to compute the gradient of the cost function, and **the BP algorithm is always a good algorithm to compute it**

Outline

1. Introduction

2. Linear Regression

3. General Machine Learning Issues

4. Single-layer Perceptrons

5. Multilayer Perceptrons
   - Definition
   - Training MLPs
   - The Backpropagation Algorithm
   - Gradient Descent with Backpropagation
   - Theoretical Results

Theoretical Results

MLPs with a single layer of hidden units have been shown to be universal approximators for several “well-behaved” classes of functions (including, for example, continuous, integrable or square integrable functions) with many families of activation functions and several metrics [Hornik et al., 1989, Leshno et al., 1993]

That means that we can, in principle, obtain an arbitrarily close approximation to the target function

In general, however, finding a neural network (with any number of hidden units) that models with precision a given training set is an NP-complete problem [Blum and Rivest, 1992, Šíma, 1994]

That's it!

Bibliography

- Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A Learning Algorithm for Boltzmann Machines. *Cognitive Science*, 9:147–169.
- Anderson, J. A. (1972). A Simple Neural Network Generating an Interactive Memory. *Mathematical Biosciences*, 14(3):197–220.
- Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, NJ.
- Blum, A. L. and Rivest, R. L. (1992). Training a 3-node Neural Network is NP-complete. *Neural Networks*, 5(1):117–127.
- Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A Training Algorithm for Optimal Margin Classifiers. In *5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152.
- Broomhead, D. S. and Lowe, D. (1988). Radial Basis Functions, Multi-variable Functional Interpolation and Adaptive Networks. Technical Report RSRE-MEMO-4148, Royal Signals and Radar Establishment Malvern (UK).
- Cortes, C. and Vapnik, V. N. (1995). Support-Vector Networks. *Machine Learning*, 20(3):273–297.
- Cristianini, N. and Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines*. Cambridge University Press, UK.
- Gauss, K. F. (1809). *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium*. Sumtibus F. Perthes et IH Besser.
- Grossberg, S. (1972). Neural Expectation: Cerebellar and Retinal Analogs of Cells Fired by Learnable or Unlearned Pattern Classes. *Kybernetik*, 10(1):49–57.
- Grossberg, S. (1980). How Does a Brain Build a Cognitive Code? *Psychological Review*, 87(1):1–51.
- Hadamard, J. (1908). Mémoire sur le Problème d'Analyse Relatif à l'Équilibre des Plaques Élastiques Encastrées. *Imprimerie Nationale*, 33.
- Hastie, T., Tibshirani, R., and Friedman, J. H. (2001). *The Elements of Statistical Learning*. Springer-Verlag, New York.

Bibliography

- Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. John Wiley & Sons.
- Hinton, G. E. (1999). Products of Experts. In *International Conference on Artificial Neural Networks*, volume 1, pages 1–6.
- Hinton, G. E. (2002). Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14:1771–1800.
- Hinton, G. E., Osindero, S., and Teh, Y. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554.
- Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507.
- Hochreiter, S. and Schmidhuber, J. (1997). Long Short-term Memory. *Neural Computation*, 9(8):1735–1780.
- Hopfield, J. (1982). Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proceedings of the National Academy of Sciences USA*, 79:2554–2558.
- Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2(5):359–366.
- Horst, R. and Tuy, H. (1993). *Global Optimization: Deterministic Approaches*. Springer-Verlag, Berlin.
- Kohonen, T. (1972). Correlation Matrix Memories. *IEEE Transactions on Computers*, C21(4):353–359.
- Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 24, pages 1106–1114.
- Lang, K. J., Waibel, A. H., and Hinton, G. E. (1990). A Time-delay Neural Network Architecture for Isolated Word Recognition. *Neural Networks*, 3(1):23–43.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551.

Bibliography

- Legendre, A. M. (1805). Nouvelles Méthodes pour la Détermination des Orbites des Comètes. *F. Didot*.
- Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function. *Neural Networks*, 6(6):861–867.
- McCulloch, W. S. and Pitts, W. (1943). A logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133.
- Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall.
- Minsky, M. L. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.
- Poggio, T. and Girosi, F. (1990). Networks for Approximation and Learning. *Proceedings of the IEEE*, 78(9):1481–1497.
- Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65(6):386–408.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Chapter 8: Learning Internal Representations by Error Propagation. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition (vol. 1)*, pages 318–362. MIT Press.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning Internal Representations Back-Propagating Errors. *Nature*, 323:533–536.
- Rumelhart, D. E. and McClelland, J. L., editors (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition (vols. 1 and 2)*. MIT Press.
- Smolensky, P. (1986). Chapter 6: Information Processing in Dynamical Systems: Foundations of Harmony Theory. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition (vol. 1)*, pages 194–281. MIT Press.
- Šíma, J. (1994). Loading Deep Networks is Hard. *Neural Computation*, 6(5):842–850.

Bibliography

- von der Malsburg, C. (1973). Self-organization of Orientation Sensitive Cells in the Striate Cortex. *Kybernetik*, 14(2):85–100.
- Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, Boston, MA.
- Widrow, B. and Hoff, M. E. (1960). Adaptive Switching Circuits. In *IRE WESCON Convention Record*.