Neural Networks and Statistical Models
A Java Implementation

Alex Foglia

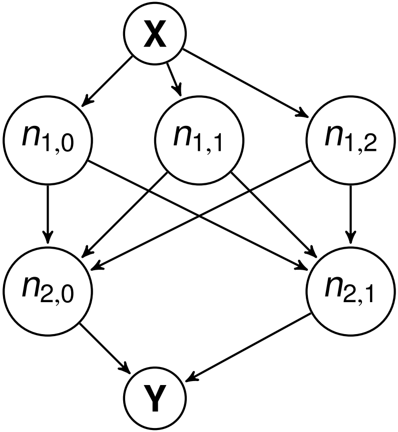
Università degli Studi di Firenze

30/01/2019

Artificial Neural Networks

- Artificial Neural Networks (NN) are a wide class of regression models, data reduction models, *nonlinear* dynamical systems
- *Computational model* composed of a set of *neurons* which aims to simulate a biological Neural Network
- Neurons consist in interconnected computing elements and they are often organized into *layers*

Example of a NN



A bit of history

- *Artifical Intelligence* (AI) is a discipline belonging to computer science which studies the theoretical foundations, the methodologies and the techniques that allow computer systems to provide services which would seem to be an exclusive pertinence to human intelligence, for the common observer. [Marco Solmavico]

A bit of history

- In the past, artificial intelligence was based on the intensive use of computing capabailities provided by computer systems
- NN is a ”novel” approach which aims, through the use of statistical models, to reproduce the concept of ”learning”
- The alleged ”intelligence” of a NN is a matter of dispute, since networks are capable of processing vast amounts of data and making prediction, only as an appropriate statistical model is able to ”fit” a sample of observed data

NN vs Statistical Models

Statistical Jargon	NN Jargon
Variables	Features
Independent Variable	Inputs
Predicted Values	Outputs
Dependent Variables	Targets
Residual	Errors
Estimation	Training, Learning
Estimation Criterion	Error Function
Observations	Patterns
Parameter Estimates	Synaptic Weights
Interactions	Higher-Order neurons
Transformations	Functional Links
Regression	Supervised Learning
Data reduction	Unsupervised Learning
Cluster Analysis	Competitive Learning
Interpolation and Extrapolation	Generalization

Terms like *sample* and *population* does not have NN equivalents, but data are often divided into a *training set* and *test set* for cross-validation.

NN and Linear Regression Model

- Linear regression models are simple statistical models used to predict the exepcted value of a dependant variable **Y** given a set of observed variables **X**
- The model is the following:

$$\mathbf{Y}_i = \beta_0 + \beta_1 \mathbf{X}_i + \epsilon_i$$

- Such a model identifies a data-fitting *line*

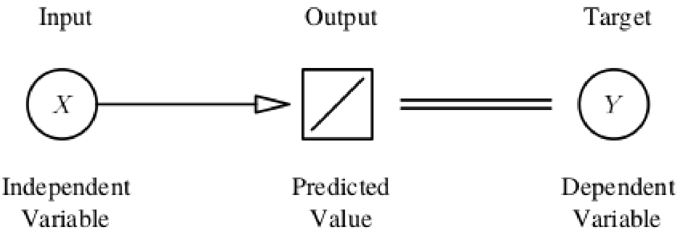
NN and Linear Regression Model

- How to estimate β_0, β_1 parameters?
- Ordinary least square method
- We define the *Loss Function S* as:

$$S(\beta_0, \beta_1) = \sum_{i=1}^{N} (y_i - \beta_0 - \beta_1 \mathbf{X}_i)^2$$

S is a function from $\mathbb{R}^2 \rightarrow \mathbb{R}$ and parameters β_0, β_1 are estimated as the arguments of S on which the function evaluates to its minima

NN and Linear Regression Model



NN and Logistic Regression Model

- A perceptron is a very small NN which usually computes a linear combination of the inputs
- A perceptron has $n > 0$ input
- Each input has a specific *weight* β_i
- The *weights* are the parameters to be estimated, moreover the parameters the model has to *learn*

NN and Logistic Regression Model

- More generally, the output of a perceptron is an evaluation of an *activation function* on the provided inputs
- Activation functions are usually *bounded*
- A bounded function maps any real input to a bounded range. Bounded activation functions are called *squashing functions*, for instance the logistic function:

$$\texttt{act}(x) = \frac{1}{1 + e^{-x}}$$

  Is a bounded activation function which maps any real argument into the range $(0, 1)$

NN and Logistic Regression Model

- What a perceptron is supposed to do is, given $\mathbf{x}$ as an input, and assuming the logistic function as the activation function:
  - Compute
  $$\hat{x} = \sum_{i=1}^{N} \beta_i \mathbf{X}_i$$
  - Return $\mathtt{act}(\hat{x}) =$
  $$\frac{1}{1 + e^{-(\beta_1 \mathbf{X}_1 + \cdots + \beta_n \mathbf{X}_n)}}$$
- Note that
$$\frac{1}{1 + e^{-(\beta_i \mathbf{X}_i + \cdots + \beta_n \mathbf{X}_n)}} = \frac{e^{(\beta_i \mathbf{X}_i + \cdots + \beta_n \mathbf{X}_n)}}{1 + e^{(\beta_i \mathbf{X}_i + \cdots + \beta_n \mathbf{X}_n)}}$$

NN and Logistic Regression Model

- The *logisitc regression model* is a *non-linear* regression model used when the dependent variable is dichotomic
- The model formula is the following:

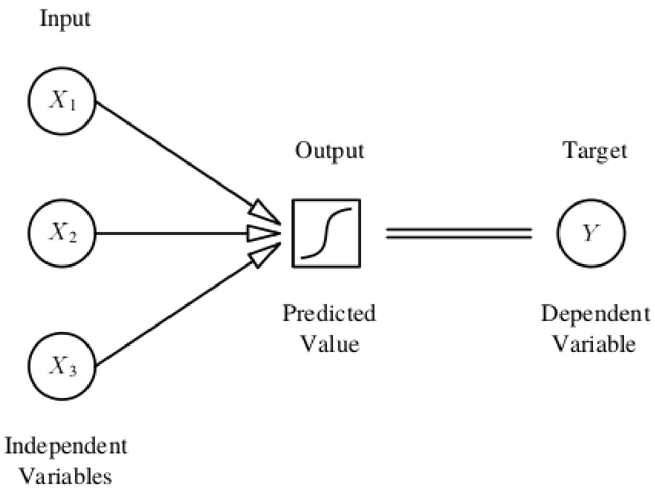
$$\mathbb{E}(\mathbf{Y}|\mathbf{X}) = \frac{e^{\beta_0+\beta_1 x_1+\cdots+\beta_n x_n}}{1+e^{\beta_0+\beta_1 x_1+\cdots+\beta_n x_n}}$$

- That is, exactly, what a perceptron with a logistic activation function computes

How to estimate β_i parameters?

- Two possible ways:
  - Maximum likelihood method
  - Gradient descent method
- The gradient descent method is an optimization algorithm which aims to estimate parameters given a set of pairs `input - expected output` (the training set)
  Aiming to minimize
  $$\sum \sum r_j^2$$
- Where the r_j is the difference between the expected output and the predicted value

NN and Logistic Regression Model

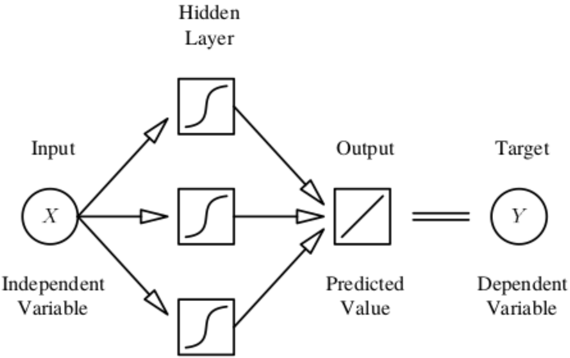


NN and Nonlinear Regression Models

- Neurons are often organized into *layers*
- NN seen before are simple perceptrons composed of two layers: the input layer and the output layer
- If it is introduced another *hidden* layer between input and output, you obtain a multi-layer perceptron (MLP)

NN and Nonlinear Regression Models

- If the model includes estimated weights between the inputs and the hidden layer, and the hidden layer uses nonlinear activation functions, the model becomes nonlinear:



NN and Nonlinear Regression Models

- This is a simple MLP implementing a non linear regression model:

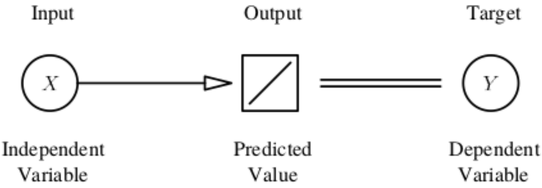
$$\mathbf{Y} = f(\mathbf{X}) + \mathbf{b}$$

- MLP are general-purpose, flexible, nonlinear models that can approximate virtually any function to any desired degree of accuracy
- MLP are called *universal approximators* and they can be used when you have little knowledge about the relationship between the independent and dependent variables

Linear Regression Implementation

- Start from the linear regression model:

$$\mathbf{Y}_i = \beta_0 + \beta_1 \mathbf{X}_i + \varepsilon_i$$



Linear Regression Implementation

- Suppose it is $\varepsilon_i = 0$
- Consider the following class:

```
public class Neuron {
  private double b0,b1;
  public double predict(double x) {
    return b0 + b1*x;
  }
  public static void main(String[] args) {
    Neuron y = new Neuron();
    System.out.println(y.predict(7));
  }
}
```

Linear Regression Implementation

- The output provided by the execution of this code is *0.0*, due to the fact that the β_0, β_1 parameters are not yet estimated.
- β_0, β_i are estimated as the minima of the function

$$S(\beta_0, \beta_1) = \sum_{i=1}^{N} (y_i - \beta_0 - \beta_1 x_i)^2$$

- It is possible to find the minima by setting $\nabla S = 0$

$$\frac{\vartheta S}{\vartheta \beta_0} = -2 \sum_{i=1}^{N} (y_i - \beta_0 - \beta_1 x_i) = 0, \quad \frac{\vartheta S}{\vartheta \beta_1} = -2 \sum_{i=1}^{N} (y_i - \beta_0 - \beta_1 x_i) x_i = 0$$

$$\beta_1 = \frac{\sigma(x, y)}{\sigma^2(x)}, \quad \beta_0 = \overline{y} - \beta_1 \overline{x}$$

Linear Regression Implementation

Extend the class code:

```
public void estimateParameters(double[] xi, double[] yi) {
  b1 = var(xi,yi) / var(xi,xi);
  b0 = mean(yi) - b1*mean(xi);
}
private double mean(double[] v) {
  double m = 0;
  for(int i=0; i<v.length;i++) {
    m+=v[i];
  }
  return m/v.length;
}
private double var(double[] x, double[] y) {
  double var = 0;
  double mx = mean(x);
  double my = mean(y);
  for(int i=0; i<x.length;i++) {
    var+= (x[i]-mx)*(y[i]-my);
  }
  return var/x.length;
}
```

Linear Regression Implementation

Suppose it exists a sample **X** for which it is known that all the values of the dependent variable **Y** are on the bisector of the second quadrant of the Cartesian plane:

$$\mathbf{X} = \{X_i\} = \{1, 2, 3, 4, 5\} \;\; \mathbf{Y} = \{Y_i\} = \{1, 2, 3, 4, 5\}$$

```java
Neuron y = new Neuron();
y.estimateParameters(new double[]{1,2,3,4,5}, new double[]{1,2,3,4,5});
System.out.println(y.predict(7));
System.out.println("Y=" + y.b0 + "+" + y.b1+"X");
```

```
7.0
Y = 0.0 + 1.0X
```

Linear Regression Implementation

- As it is expected, even if in the *training set* does not appear the expected output for value 7, our linear regression model is able to predict the output for such a value: *7.0*
- In fact the regression line is exactly the requested identity function

Extending the code: Logistic Regression

The code must be modified in order to implement a logistic regression model:

```java
public class Neuron {
  private double[] weights;

  public Neuron(int n_inputs) {
    weights = new double[n_inputs];
  }

  private double logistic(double x) {
    return 1 / (1 + Math.exp(-x));
  }

  public double predict(double[] inputs) {
    double sum = 0;
    for (int i = 0; i < inputs.length; i++) {
      sum += weights[i] * inputs[i];
    }
    return logistic(sum);
  }
}
```

Extending the code: Logistic Regression

Now the job is to modify the `estimateParameter()` method seen before in order to estimate weights using the gradient descent method:

```
public void estimateParameters(double[][] xi, double[] yi) {
  double[] gradient = new double[weights.length];
  for (int i = 0; i < xi.length; i++) {
    for (int j = 0; j < xi[0].length; j++) {
        gradient[j] += xi[i][j] * (yi[i] - predict(xi[i]));
    }
  }
  for (int j = 0; j < weights.length; j++)
    weights[j] += gradient[j];
}
```

Note that we are using a particular Loss Function for which it is possible to compute the gradient in such an easy way

Extending the code: Logistic Regression

Consider this piece of python code:

```
import math
b0 = −1
b1 = −2
squash = lambda x: math.exp((b0*x[0]+b1*x[1]))/(1+math.exp((b0*x[0]+b1*x[1])))
print squash([1,0])
print squash([0,1])
print squash([0,0])
print squash([1,1])
```

Executing it, it is possible to generate data from the logistic function:

$$f(\mathbf{x}) = \frac{e^{-x_1-2x_2}}{1+e^{-x_1-2x_2}},$$

Extending the code: Logistic Regression

It is possible now to train this simple network with such data:

```
Neuron n = new Neuron(2);
double[][] in = {{1,0},{0,1},{0,0},{1,1}};
double[] out = {0.2689414213699951,0.11920292202211755,0.5,0.04742587317756679};
for(int i=0; i<10000;i++)
  n.estimateParameters(in, out);
System.out.println(Arrays.toString(n.weights));
```

Obtaining the following coefficients:

```
[-1.0000000000000004, -1.9999999999999993]
```

Which are approximately the β_0, β_1 used to generate data

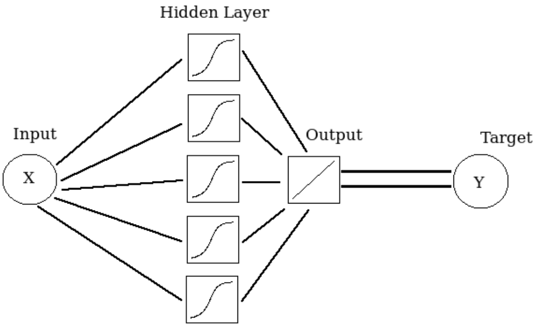
The XOR operator

Consider the following *truth table*:

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

This is known as the logical operator *xor*. A NN composed of a nonlinear MLP is capable to learn the *xor* computation.

From Logistic Regression to Universal Approximators



This is a MLP, or *universal approximator*. The aim is to modify the provided code in order to implement such a MLP able to learn to compute the *xor*.

From Logistic Regression to Universal Approximators

Start from the Neuron class:

```
public class Neuron {
  public double[] weights;
  public double[] inputs;
  public double output;
  public Neuron(int n_inputs) {
    weights = new double[n_inputs];
    for(int i = 0; i < n_inputs; i++){
      weights[i] = Math.random();
    }
  }
  private double logistic(double x) {
    return 1/(1+Math.exp(-x));
  }
  public double predict(double[] inputs) {
    this.inputs = inputs;
    double sum = 0;
    for(int i=0; i<inputs.length;i++) {
      sum += inputs[i] * weights[i];
    }
    this.output = logistic(sum);
    return output;
  }
}
```

From Logistic Regression to Universal Approximators

Since MLPs have more than one layer, we define, from a programming point of view, what a layer is: a collection of Neurons.

```java
public class NeuronLayer {
  public Neuron[] neurons;
  public NeuronLayer(int n, int inputsPerNeuron) {
  this.neurons = new Neuron[n];
  for (int i = 0; i < n; i++) {
    this.neurons[i] = (new Neuron(inputsPerNeuron));
  }
  }
  public double[] feedForward(double[] inputs) {
    double[] outputs = new double[neurons.length];
    for (int i = 0; i < neurons.length; i++) {
      outputs[i] = (neurons[i].predict(inputs));
    }
    return outputs;
  }
  public double[] getOutputs() {
    double[] outputs = new double[neurons.length];
    for (int i = 0; i < neurons.length; i++) {
      outputs[i] = neurons[i].output;
    }
  return outputs;
  }
}
```

From Logistic Regression to Universal Approximators

Putting layers together means to construct a MLP Neural Network:

```
public class NeuralNetwork {
    private int n_in, n_hid, n_out;
    private NeuronLayer hidden, output;
    public NeuralNetwork(int inputNeurons, int hiddenNeurons, int numberOfOutputs){
      n_in = inputNeurons;
      n_hid = hiddenNeurons;
      n_out = numberOfOutputs;
      hidden = new NeuronLayer(n_hid, n_in);
      output = new NeuronLayer(n_out, n_hid);
    }
  }
}
```

The MLP must be capable of performing prediction:

```
public double[] feedForward(double[] inputs) {
  double[] hidden_outputs = hidden.feedForward(inputs);
  return output.feedForward(hidden_outputs);
}
```

Backpropagation

Last, the MLP must be capable of being trained in order to estimate neurons weights:

```
public void train(double[] training_in, double[] training_out) {
  feedForward(training_in);
  double[] deltaWrtOut = new double[n_out];
  for (int i = 0; i < n_out; i++) {
    double target_output = training_out[i];
    double actual_output = output.neurons[i].output;
    double deltaWrtInput = -(target_output - actual_output) * actual_output * (1 -
      actual_output);
    deltaWrtOut[i] = deltaWrtInput;
  }
```

```
  double[] deltaWrtHid = new double[n_hid];
  for(int i=0; i<n_hid;i++) {
    double deltaWrtHiddenOut = 0;
    for(int j=0; j<n_out;j++) {
      deltaWrtHiddenOut+=deltaWrtOut[j] * output.neurons[j].weights[i];
    }
  double actual_output = hidden.neurons[i].output;
  double deltaWrtIn = actual_output * (1 - actual_output);
  deltaWrtHid[i] = deltaWrtHiddenOut * deltaWrtIn;
  }
  for (int i = 0; i < n_out; i++) {
    for (int j = 0; j < n_hid; j++) {
      double act_input = output.neurons[i].inputs[j];
      double deltaWrtWeight = deltaWrtOut[i] * act_input;
      output.neurons[i].weights[j] -= deltaWrtWeight;
    }
  }
  for (int i = 0; i < n_hid; i++) {
    for (int j = 0; j < n_in; j++) {
      double act_input = hidden.neurons[i].inputs[j];
      double deltaWrtWeight = deltaWrtHid[i] * act_input;
      hidden.neurons[i].weights[j] -= deltaWrtWeight;
    }
   }
}
```

Error Evaluation

Another method is added in order to compute the total network error with respect to a training set:

```java
public double totalError(double[][][] training_sets) {
  double err = 0;
  for (int i = 0; i < training_sets.length; i++) {
    double[] t_in = training_sets[i][0];
    double[] t_out = training_sets[i][1];
    double[] act_out = feedForward(t_in);
    for (int j = 0; j < act_out.length; j++) {
      double target_output = t_out[j];
      double actual_output = output.neurons[j].output;
      double squareError = 0.5 * Math.pow(target_output - actual_output, 2);
      err += squareError;
    }
  }
return err;
}
```

Launch the NN

Now the model is ready to use. Start from constructing the model shown before. It has 2 inputs, 5 hidden neurons, and 1 output:

```
NeuralNetwork nn = new NeuralNetwork(2, 5, 1);
```

Define the training sets as defined in the *xor* table of truth:

```
double[][][] training_sets = {{{0, 0}, {0}},{{0, 1}, {1}},{{1, 0}, {1}},{{1, 1}, {0}}};
```

Train the network:

```
System.out.println("Error before training: "+nn.totalError(training_sets));
for (int i = 0; i < 10000; i++) {
  int randIndex = (int) (Math.random() * training_sets.length);
  double[] t_in = training_sets[randIndex][0];
  double[] t_out = training_sets[randIndex][1];
  nn.train(t_in, t_out);
}
System.out.println("Error after training: "+nn.totalError(training_sets));
```

And this is the output:

```
Error before training: 0.7825167086789118
Error after training: 0.0019131802708672071
```

Launch the NN

- What it is implemented is a MLP which performs a *xor* approximation using the statistical model *non-linear regression*
- If you try to train the MLP with other datasets which arise from considering other functions than the *xor*, such as the *and*, *nand*, etc, the MLP will approximate them with the same degree of accuracy
- The entire code is available at: https://github.com/alexfoglia1/MASL/tree/master/exam/JavaNN/src