A Gentle and Practical Introduction to Neural Networks

Soufiane Belharbi

soufiane.belharbi@insa-rouen.fr

LITIS lab., Apprentissage team - INSA de Rouen, France
ASI 4

March, 15th, 2017

Plan

Results

When you finish this course you will:

- Understand what is a neural network.
- How to train it.

By doing the **Hands on Lab 1**, you will be able to:

- Implement a neural network using Keras.
- Implement a neural network using Theano.

Machine Learning

What is machine learning (ML)?

ML is programming computers (algorithms) to optimize a performance criterion using **example data or past experience**.

Learning a task

Learn general models from data to perform a specific task f.

$$f_{\mathbf{w}} : \mathbf{x} \longrightarrow \mathbf{y}$$

$\mathbf{x}$: input
$\mathbf{y}$: output (target, label)
$\mathbf{w}$: parameters of f
$f(\mathbf{x}; \mathbf{w}) = \mathbf{y}$

From training to predicting the future: Learn to predict

1. Train the model using data examples $(\mathbf{x}, \mathbf{y})$
2. Predict the $\mathbf{y}_{new}$ for the new coming $\mathbf{x}_{new}$

Machine Learning

What is machine learning (ML)?

ML is programming computers (algorithms) to optimize a performance criterion using **example data or past experience.**

Learning a task

Learn general models from data to perform a specific task f.

$$f_{\mathbf{w}} : \mathbf{x} \longrightarrow \mathbf{y}$$

$\mathbf{x}$: input
$\mathbf{y}$: output (target, label)
$\mathbf{w}$: parameters of f
$f(\mathbf{x}; \mathbf{w}) = \mathbf{y}$

From training to predicting the future: **Learn to predict**

1. Train the model using data examples $(\mathbf{x}, \mathbf{y})$
2. Predict the $\mathbf{y}_{new}$ for the new coming $\mathbf{x}_{new}$

Machine Learning

What is machine learning (ML)?

ML is programming computers (algorithms) to optimize a performance criterion using **example data or past experience**.

Learning a task

Learn general models from data to perform a specific task f.

$$f_{\mathbf{w}} : \mathbf{x} \longrightarrow \mathbf{y}$$

$\mathbf{x}$: input
$\mathbf{y}$: output (target, label)
$\mathbf{w}$: parameters of f
$f(\mathbf{x}; \mathbf{w}) = \mathbf{y}$

From training to predicting the future: **Learn to predict**

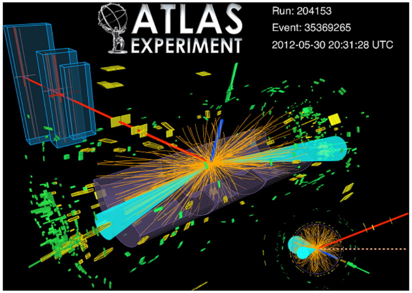
1. Train the model using data examples $(\mathbf{x}, \mathbf{y})$
2. Predict the $\mathbf{y}_{new}$ for the new coming $\mathbf{x}_{new}$

Machine Learning: Applications

- Face detection/recognition
- Image classification
- Handwriting recognition(postal address recognition, signature verification, writer verification, historical document analysis (DocExplore `http://www.docexplore.eu`))
- Speech recognition, Voice synthesizing
- Natural language processing (sentiment/intent analysis, statistical machine translation, Question answering (Watson), Text understanding/summarizing, text generation)
- Anti-virus, anti-spam
- Weather forecast
- Fraud detection at banks
- Mail targeting/advertising
- Pricing insurance premiums
- Predicting house prices in real estate companies
- Win-tasting ratings
- Self-driving cars, Autonomous robots
- Factory Maintenance diagnostics
- Developing pharmaceutical drugs (combinatorial chemistry)
- Predicting tastes in music (Pandora)
- Predicting tastes in movies/shows (Netflix)
- Search engines (Google)
- Predicting interests (Facebook)
- Web exploring (sites like this one)
- Biometrics (finger prints, iris)
- Medical analysis (image segmentation, disease detection from symptoms)
- Advertisements/Recommendations engines, predicting other books/products you may like (Amazon)
- Computational neuroscience, bioinformatics/computational biology, genetics
- Content (image, video, text) categorization
- Suspicious activity detection
- Frequent pattern mining (super-market)
- Satellite/astronomical image analysis

ML in Physics

Event detection at CERN (The European Organization for Nuclear Research)

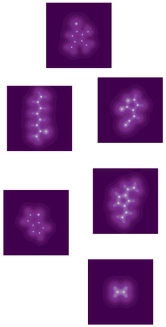


⇒ Use ML models to determine the probability of the event being of interest.

⇒ **Higgs Boson Machine Learning Challenge** (`https://www.kaggle.com/c/higgs-boson`)

ML in Quantum Chemistry

Computing the electronic density of a molecule
$\Rightarrow$ Instead of using physics laws, use ML (**FAST**).



See Stéphane Mallat et al. work: `https://matthewhirn.files.wordpress.com/2016/01/hirn_pasc15.pdf`

How to model $f_{\mathbf{w}}$?

Models

- Parametric ($\mathbf{w}$) vs. non-parametric
- Parametric ($\mathbf{w}$): how to find $\mathbf{w}$ $\rightarrow$ **train** the model using **data**
- Training: supervised (use $(\mathbf{x}, \mathbf{y})$) vs. unsupervised (use only $\mathbf{x}$)
- Training = optimizing an **objective cost**

Different models to learn $f_{\mathbf{w}}$

- Kernel models (support vector machine (SVM))
- Decision tree
- Random forest
- Linear regression
- K-nearest neighbor
- Graphical models
  - Bayesian networks
  - Hidden Markov Models (HMM)
  - Conditional Random Fields (CRF)
- Neural networks (Deep learning): DNN, CNN, RBM, DBN, RNN.

How to model $f_{\mathbf{w}}$?

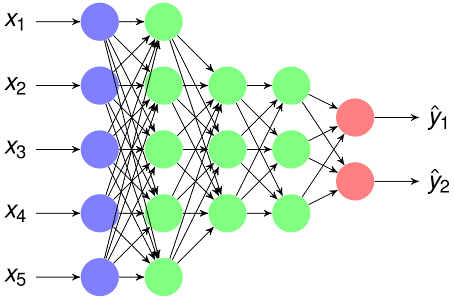
Models

- Parametric ($\mathbf{w}$) vs. non-parametric
- Parametric ($\mathbf{w}$): how to find $\mathbf{w}$ $\rightarrow$ **train** the model using **data**
- Training: supervised (use $(\mathbf{x}, \mathbf{y})$) vs. unsupervised (use only $\mathbf{x}$)
- Training = optimizing an **objective cost**

Different models to learn $f_{\mathbf{w}}$

- Kernel models (support vector machine (SVM))
- Decision tree
- Random forest
- Linear regression
- K-nearest neighbor
- Graphical models
  - Bayesian networks
  - Hidden Markov Models (HMM)
  - Conditional Random Fields (CRF)
- Neural networks (Deep learning): DNN, CNN, RBM, DBN, RNN.

Feed-Forward Neural Networks



- State of the art in many task: computer vision, natual language processing.
- Training requires large data
- To speed up the training: use GPUs cards
- Training **deep** neural networks is **difficult**

  $\Rightarrow$ Vanishing gradient

  $\Rightarrow$ More parameters $\Rightarrow$ Need more data

Linear Regression

To perform a regression, $\mathbf{y} = f(\mathbf{x}) \in \mathcal{R}$, $\mathbf{x} \in \mathcal{R}d$:

$$f(\mathbf{x}) = w^T\mathbf{x} + b =< W^T, \mathbf{x} > +b = \sum_{i=1}^{d} w_i x_i + b$$

$$=< [W^T; b], [\mathbf{x}; 1] >= \sum_{i=1}^{d+1} w_i x_i$$

$b \sim \epsilon$ (noise, bias). Set of parameters: $\mathbf{W} = \{W, b\}$.
Question: given the input feature space is in $\mathcal{R}^{100}$. What is the size of the parameter w?

Logistic Regression

Generalization toward discrete output (binary classification)

$$f(\mathbf{x}) = sigm(W^t\mathbf{x} + b)$$

$$sigm(\eta) = \frac{1}{1 + exp(-\eta)} \in [0, 1]$$

Decision rule: simple thresholding.

$$p(y = 1|x, \mathbf{W}) = sigm(W^T x + b)$$

$$y = 1 \Leftrightarrow p(y = 1|x) > 0.5$$

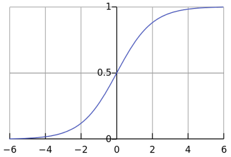


Figure 1: Standard logistic sigmoid function

Softmax Function

A generalization of the logistic function to multi-class:
i.e: concatenation of a lot of logistic function + normalization as follows:

$$\sigma(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad for j = 1, \ldots, K.$$

$$P(y = j|x) = \frac{e^{W_j^T x}}{\sum_{k=1}^{K} e^{W_k^T x}}$$

Interpret the output as a **probabilities.**

One Neuron (Unit)

$$
\begin{aligned}
g(\mathbf{x}) &= \varphi(W^t\mathbf{x} + b) \\
\varphi(\eta) &= \text{non linear transformation (activation function)} \\
&= \text{: monotonic, continuously differentiable}
\end{aligned}
$$

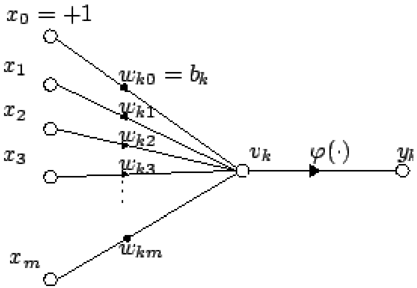


Figure 2: A simple artificial neuron

A Layer of Neurons

A layer is a set of simple neurons. It transforms $x \in \mathcal{R}^{d} \rightarrow x' \in \mathcal{R}^{d'}$.
d: **input dimension**. d': **output dimension**.
The parameters of a layer: is matrix $W \in \mathcal{R}^{d \times d'}$ and a vector of biases $b \in \mathcal{R}^{d'}$. Each column of W and the correspondent component in b is the parameters of one neuron (vector and bias).
The neurons are usually have the same activation function. Therefore, a layer can be written as:

$$g(x) = \varphi(Wx + b)$$

φ is a **piece-wise operator**.
For the sake of notation/tradition, we usually write: $Wx + b$. In practice, the implementation is done as: $xW + b$ where: ($W \in \mathcal{R}^{d \times d'}$).
Think of a layer as a **block** with an input (x) and an output $g(x)$.
We usually refer to this type of layers by: **dense layer** or **fully connected layer**.
A layer attempts to **learn** a transformation (a new **representation**) of its input.
Notation: W: weights. b: biases.

Matrix Computation

Think matrix:

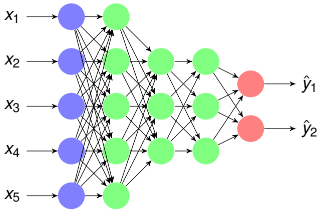
- All the operations are done using matrices and piece-wise operators.
- Process many samples at once: The input of a layer can be a matrix, where each row is a sample (vector):

$$g(x) = \varphi(Wx + b)$$

In this case, the output ($g(x)$) of the layer is a matrix also, where each row is a vector which corresponds to an input vector.

Multilayer Network

Multilayer network (a.k.a: multilayer perceptron: MLP) is a cascade of layer where the input of a layer is the output of the previous one. (**nested transformation**)



The output of this 4 layers network for classification can be written as one single function:

$$f(x) = y = softmax(\underbrace{W_4\,\underbrace{\varphi_3(W_3\,\underbrace{\varphi_2(W_2\,\underbrace{\varphi_1(W_1 x + b_1)}_{\text{first hidden layer}} + b_2)}_{\text{second hidden layer}} + b_3)}_{\text{third hidden layer}} + b_4)}_{outputlayer}$$

List of the parameters of this network: $[W_1, b_1, W_2, b_2, W_3, b_3, W_4, b_4]$.
Notations:

- **Input layer**: The input representation (x). (blue) (no parameters)
- **Output layer**: The last layer (the one with the output = $y = f(x)$). (red one)
- **Hidden layer**: The layer which is not input nor output. (green ones).

On the Activation Functions

Most used activation (non-linearity) functions:

- Softmax $\in [0, 1]$
- Sigmoid $\in [0, 1]$
- Hyperbolic Tangent $\in [-1, 1]$
- Rectified Linear Unit (Relu): $max(0, x)$ which means

$$Relu(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

You can mix different activations in the same network.
If the network requires to output values $\in [-1, 1]$ it is better to use the Hyperbolic tangent function at the output.

General View

- Supervised data: Pair of samples (x, y). $x \in \mathcal{R}^d$. Classification: y is categorical then converted to exclusive binary vectors. Regression: $y \in \mathcal{R}^{d'}$.
- Specify the training criterion (objective loss, error function): what do you want the network to learn?
- Specify the architecture of the network (#layer, #neurons per layer, . . .). Initialize the network.
- Training (Learning): Search in the space parameters using **Stochastic Gradient Descent** (SGD), **back-propagation error**.
- Stop training when reaching the optimum. The network is ready to be tested.

On the initialization

The weights are usually initialized randomly following some heuristics. The biases are usually set to zero.
For example, for a layer with size input *in* and size of output *out*:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{in + out}}, \frac{\sqrt{6}}{\sqrt{in + out}}\right]$$

$U[-a, a]$ is the uniform distribution in the interval $(-a, a)$.

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks."

Aistats. Vol. 9. 2010.

Loss Function

Consider the network as a parametric function.
Specify the training criterion to **minimize**.
Supervised data: $\mathcal{D} = \{(x_i, y_i), i = 1, \dots, N\}$. $f(x)$ network output.
Set of all the network parameters: $\mathbf{W} = \{W_1, b_1, W_2, b_2, \dots\}$.

- **Regression**: Using mean squared errors (MSE)

$$\mathcal{J}(\mathcal{D}; \mathbf{W}) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} ||f(x_i; \mathbf{W}) - y_i||^2$$

- **Binary classification**: Using cross-entropy (negative log-likelihood (NLL))

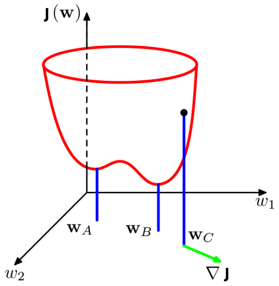
$$\mathcal{J}(\mathcal{D}; \mathbf{W}) = \frac{-1}{N} \sum_{i=1}^{N} y_i log(f(x_i; \mathbf{W})) + (1 - y_i) log(1 - f(x_i; \mathbf{W})) \quad y_i \in \{0, 1\}$$

- **Multi-classification**: multi-class NLL

$$\mathcal{J}(\mathcal{D}; \mathbf{W}) = \frac{-1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} log(f(x_i; \mathbf{W})_k)$$

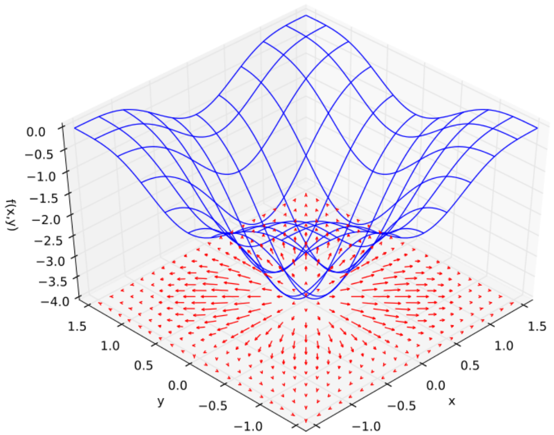
Parameter Optimization

$\mathcal{J}$ is **non-convex**.



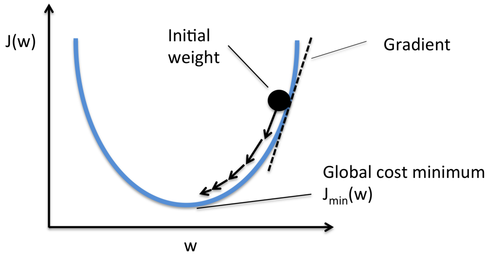
Goal: search the parameters where: $\nabla \mathcal{J}(\mathbf{W}) = 0$ **(gradient)**
$\nabla \mathcal{J}(\mathbf{W}) = \frac{\partial f(\mathbf{W})}{\partial \mathbf{W}}$ the **gradient** is a **vector-valued function**.
Problem: stationary points: minima, maxima, saddle points. Find a **global minima** (smallest value of the error).
Use iterative numerical methods (for example the gradient information).

Parameter Optimization: Gradient Descent methods



The gradient points in the direction of the greatest rate of increase of the error function. But, we want to decrease it (go the opposite direction).

Parameter Optimization: Gradient Descent methods

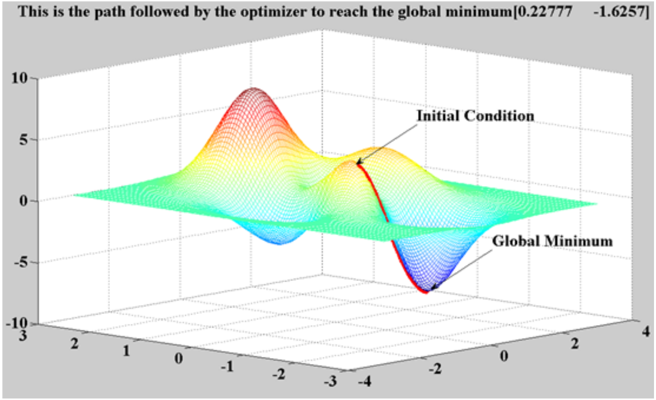


$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \epsilon \frac{\partial \mathcal{J}(\mathcal{D};\mathbf{w})}{\partial \mathbf{w}}$

$\epsilon \in \mathcal{R}^+$ is a learning rate (speed: how much take information from the current error.). Usually it is very small (order of 10^{-3}). Small values: go slow. Larger values: go fast. The more we approach to the optimum solution, the more we need to go slower (valley oscillation).

This is the simplest SGD (i.e. use a **global learning rate**). Do the parameters (layers) learn in the same speed?

Parameter Optimization: Gradient Descent methods



$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \epsilon \frac{\partial \mathcal{J}(\mathcal{D};\mathbf{w})}{\partial \mathbf{w}}$$

How to Calculate the Gradient in Neural Networks? Back-propagation Error

Tradition:

1. Forward pass: Forward x into the network (i.e. calculate $f(x)$) in order to calculate the output/activation of each neuron and calculate the loss function given the current parameters.
2. Back-ward pass: Calculate the gradient of the loss function with respect to each parameter using the **chain rule**.

Intuition:

1. Forward pass: Ask the network to predict an output given an input. Track the state of every neuron in the network. Compare the prediction to the ground truth.
2. Back-ward pass: Compute how much each neurone participate in committing the total error.

Issues: vanishing gradient (for deeper networks), slow learning, ...

How to Calculate the Gradient in Neural Networks? Back-propagation Error

Nowadays:
Use automatic differentiation over graphs. One simple call: `grad(loss=`$\mathcal{J}$`,` $w\_r\_t = \mathbf{W}$`)` to obtain the list of all the gradient expressions with respect to each parameter. Then evaluate the gradient using x and $\mathbf{W}$.

Pseudo-code of Training a Neural Network

Algorithm 1 Gradient Descent

```
 1: $\mathcal{D}$ is the *shuffled* training set. $\epsilon = 10^{-3}$
 2: **while** True **do**
 3:     loss = $\mathcal{J}(\mathcal{D}, \mathbf{W})$
 4:     d_loss_wrt_params = ... # compute the gradient
 5:     $\mathbf{W} = \mathbf{W} - \epsilon *$ d_loss_wrt_params
 6:     After some times, start reducing $\epsilon$
 7:     **if** <stopping condition is met> **then**
 8:         return $\mathbf{W}$
 9:     **end if**
10: **end while**
```

Pseudo-code of Training a Neural Network

Algorithm 2 Stochastic Gradient Descent (online training)

```
 1: $\mathcal{D}$ is the *shuffled* training set. $\epsilon = 10^{-3}$
 2: for $(x_i, y_i) \in \mathcal{D}$ do
 3:     loss = $\mathcal{J}((x_i, x_j), \mathbf{W})$
 4:     d_loss_wrt_params = . . . # compute the gradient
 5:     $\mathbf{W} = \mathbf{W} - \epsilon *$ d_loss_wrt_params
 6:     After some times, start reducing $\epsilon$
 7:     if <stopping condition is met> then
 8:         return $\mathbf{W}$
 9:     end if
10: end for
```

Pseudo-code of Training a Neural Network

Algorithm 3 Stochastic Gradient Descent (works better)

```
 1: $\mathcal{D}$ is the *shuffled* training set. Split $\mathcal{D}$ into mini-batches(sets)
    $B$. $\epsilon = 10^{-3}$.
 2: for $B \in \mathcal{D}$ do
 3:     loss = $\mathcal{J}(B, \mathbf{W})$
 4:     d_loss_wrt_params = ... # compute the gradient
 5:     $\mathbf{W} = \mathbf{W} - \epsilon *$ d_loss_wrt_params
 6:     After some times, start reducing $\epsilon$
 7:     if <stopping condition is met> then
 8:         return $\mathbf{W}$
 9:     end if
10: end for
```

Find a tradoff in the choice of the size of the mini-batch (training time, performance).

Regularization

Under-fitting, fitting, over-fitting: find a compromise between the complexity of the model (number of parameters) and the size of training data.

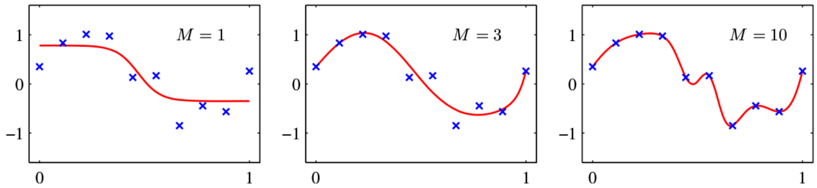


Figure 3: A 2-layer network trained with 10 points using a hidden layer with 1, 3 and 10 neurons.

Regularization (2)

Search (while training) **smooth solutions**:

$$\mathcal{J}(\mathcal{D}, \mathbf{W}) = \sum_{i=1}^{N} \mathcal{C}(f(x_i; \mathbf{W}), y_i) + \lambda \phi(\mathbf{W})$$

$\mathcal{C}(.,.)$ is a cost function.

- L_1 regularization: $\phi(\mathbf{W}) = ||\mathbf{W}||_1$
- L_2 regularization: $\phi(\mathbf{W}) = \frac{1}{2}||\mathbf{W}||_2^2$

Named by **weight decay.** λ regularization coefficient (order of 10^{-4}) to indicate how much to consider the regularization importance. (performed usually only over the weights not the biases).

Regularization (3)

Training with input transformation. (also data augmentation)

$$\mathcal{J}(\mathcal{D}, \mathbf{W}) = \sum_{i=1}^{N} \mathcal{C}(f(x_i; \mathbf{W}), y_i) + \mathcal{C}(f(\psi(x_i); \mathbf{W}), y_i)$$

$\psi(.)$ is a transformation function. For example: adding random noise to x.

Regularization (4)

Stop early before you over-fit (early stopping).
This requires an extra dataset (validation) in order to evaluate the performance of the model while training. The training is stopped when the measured error increases over the validation set.

Algorithm 4 Stochastic Gradient Descent (works better)

```
1:  $\mathcal{D}$ is the *shuffled* training set. Split $\mathcal{D}$ into mini-batches(sets)
    $B$. $\epsilon = 10^{-3}$.
2:  for $B \in \mathcal{D}$ do
3:      loss = $\mathcal{J}(B, \mathbf{W})$
4:      d_loss_wrt_params = ... # compute the gradient
5:      $\mathbf{W} = \mathbf{W} - \epsilon *$ d_loss_wrt_params
6:      if it is time to validate then
7:          evaluate the model over the validation set.
8:          if Performance improved then
9:              Save the current $\mathbf{W}$
10:         end if
11:     end if
12:     After some times, start reducing $\epsilon$
13:     if <stopping condition is met> then
14:         return the best saved $\mathbf{W}$
15:     end if
16: end for
```

Few Tricks

- Normalized the input data. (advanced: normalize the internal activations).
- Shuffle the training data during training (each epoch) (avoid periodicity).
- Be very careful with the learning rate. Start always small and keep it small. Going faster does not mean better.

- Neural networks are good at approximating any continuous function.
- They have impressive performance.
- Difficult machines to train.
- To well train them you need a lot of data.
- Training needs speed (GPUs).
- A lot of tricks. (sorcery? not exactly, but needs experience.)

Questions

Thank you for your attention,

Questions?

Is that all?
Yes.
What now?
Hands on Lab.

Hands on Lab

Break for 15 minutes.

....

1. Implement a neural network using Keras.
2. Implement your own neural network using Theano.