Artificial Neural Networks

Ronan Collobert
ronan@collobert.com
Introduction: Neural Networks in 1980

Figure 2 Hierarchical neural network structure
Introduction: Neural Networks in 2011

Stack matrix-vector multiplications interleaved with non-linearity

- Where does this come from?
- How to train them?
- Why does it generalize?
- What about real-life inputs (other than vectors $x$)?
- Any applications?
Dendrites connected to other neurons through synapses
Excitatory and inhibitory signals are integrated
If stimulus reaches a threshold, the neuron fires along the axon
Neuron as linear threshold units

- Binary inputs $x \in \{0, 1\}^d$, binary output, vector of weights $w \in \mathbb{R}^d$

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x > T \\ 0 & \text{otherwise} \end{cases}$$

- A unit can perform OR and AND operations
- Combine these units to represent any boolean function
- How to train them?
Perceptron: Rosenblatt (1957)

- **Input**: retina $x \in \mathbb{R}^n$
- **Associative area**: any kind of (fixed) function $\varphi(x) \in \mathbb{R}^d$
- **Decision function**:
  \[
  f(x) = \begin{cases} 
  1 & \text{if } w \cdot \varphi(x) > 0 \\
  -1 & \text{otherwise}
  \end{cases}
  \]

- **Training**: minimize $\sum_t \max(0, -y^t w^t \cdot \varphi(x^t))$, given $(x^t, y^t) \in \mathbb{R}^d \times \{-1, 1\}$
  \[
  w^{t+1} = w^t + \begin{cases} 
  y^t \varphi(x^t) & \text{if } y^t w \cdot \varphi(x^t) \leq 0 \\
  0 & \text{otherwise}
  \end{cases}
  \]
Perceptron: Convergence (Novikoff, 1962)

- Cauchy-Schwarz ($\rho_{\text{max}} \triangleq 2/||u||$)...
  \[ u \cdot w^t \leq ||u|| ||w^t|| \leq \frac{2}{\rho_{\text{max}}} ||w^t|| \]

- $u$ defines maximum margin separating hyperplane...
  \[ u \cdot w^t = u \cdot w^{t-1} + y^t u \cdot x^t \geq u \cdot w^{t-1} + 1 \geq t \]

- When we do a “mistake”...
  \[ ||w^t||^2 = ||w^{t-1}||^2 + 2y^t w^{t-1} \cdot x^t + ||x^t||^2 \leq ||w^{t-1}||^2 + R^2 \leq t R^2 \]

- We get:
  \[ t \leq \frac{4 R^2}{\rho_{\text{max}}^2} \]
Adaline: Widrow & Hoff (1960)

- Problems of the Perceptron:
  - Separable case: does not find a hyperplane equidistant from the two classes
  - Non-separable case: does not converge

- Adaline (Widrow & Hoff, 1960) minimizes

\[
\frac{1}{2} \sum_{t} (y^t - w^t \cdot \varphi(x^t))^2
\]

- Delta rule:

\[
w^{t+1} = w^t + \lambda (y^t - w^t \cdot x^t) x^t
\]
**Perceptron: Margin**

See (Duda & Hart, 1973), (Krauth & Mézard, 1987), (Collobert, 2004)

- **Poor generalization** capabilities in practice
- No control on the **margin**:

\[
\rho = \frac{2}{||w^T||} \geq \frac{\rho_{\text{max}}}{R^2}
\]

- **Margin Perceptron**: minimize \( \sum_t \max(0, 1 - y^t w^t \cdot \varphi(x^t)) \)

\[
w^{t+1} = w^t + \lambda \begin{cases} 
  y^t \varphi(x^t) & \text{if } y^t w^t \cdot \varphi(x^t) \leq 1 \\
  0 & \text{otherwise}
\end{cases}
\]

- Finite number of updates:

\[
t \leq \frac{4}{\rho_{\text{max}}^2}(\frac{2}{\lambda} + R^2)
\]

- **Control** on the margin:

\[
\rho \geq \rho_{\text{max}} \frac{1}{2 + R^2 \lambda}
\]
Perceptron: In Practice

Original Perceptron (10/40/60 iter)

Margin Perceptron (10/120/2000 iter)
Regularization

- In many machine-learning algorithms (including SVMs!) early stopping (on a validation set) is a good idea

From (Prechelt, 1997)

- Weight decay

\[ \mu \| w \|^2 + \max(0, 1 - y^t w^t \cdot \varphi(x^t)) \]

This is the SVM cost!
Consider the decision function

\[ f(x) = \begin{cases} 
1 & \text{if } w \cdot \varphi(x) > 0 \\
-1 & \text{otherwise}
\end{cases} \]

- Non-linearity achieved by hand-crafting a non-linear \( \varphi(\cdot) \)

Here \( \varphi(x) = \varphi(x_1, x_2) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2) \)

- Problem: the dot product is slow to compute in high dimensions
See (Aizerman, Braverman and Rozonoer, 1964)

Consider now the update

\[ w^{t+1} = w^t + \begin{cases} y^t \varphi(x^t) & \text{if } y^tw \cdot \varphi(x^t) \leq 0 \\ 0 & \text{otherwise} \end{cases} \]

Decision function at the \( t^{th} \) example can be written as:

\[ f^t(x) = \sum_{t \in \text{“updated”}} y^t \varphi(x^t) \cdot \varphi(x) \]

Can use a kernel instead

\[ K(x, x^t) = \varphi(x^t) \cdot \varphi(x) \]

E.g., for \( \varphi(x) = \varphi(x_1, x_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2) \) a possible kernel is

\[ K(x, x^t) = (x \cdot x^t)^2 \]

\( K(\cdot, \cdot) \) is a kernel if \( \forall g \)

such that \( \int g(x)^2 dx < \infty \) then \( \int K(x, y) g(x) g(y) dx dy \geq 0 \)
Support Vector Machines unify nicely all the previous concepts

- Early versions: (Vapnik & Lerner, 1963), (Vapnik, 1979)
  - Perceptron + Margin + Regularization
    = soft-margin Support Vector Machines
      (Cortes & Vapnik, 1995)
  - Perceptron + Margin + Regularization + Kernel
    = non-linear Support Vector Machines
      (Hard-margin SVMs: Boser, Guyon & Vapnik, 1992)

For linear SVM, **primal** optimization is ok
For non-linear kernels, sparsity issues with gradient descent in the primal
  $\rightarrow$ efficient algorithms exist in the **dual**, or consider a **budget**
see SVM course
How to *train* a “good” $\varphi(\cdot)$?

Neocognitron: (Fukushima, 1980)
Going Non-Linear: Adding Layers

- Madaline: (Winter & Widrow, 1988)

![Layered feed-forward ADALINE network.](image)

- Multi-Layer Perceptron

\[ x \xrightarrow{W^1} \cdot \xrightarrow{\tanh(\cdot)} \xrightarrow{W^2} \text{score} \]

- Training solution: gradient descent
Universal Approximator (Cybenko, 1989)

- Any function
  \[ g : \mathbb{R}^d \rightarrow \mathbb{R} \]
  can be approximated (on a compact) by a **two-layer neural network**

\[ x \xrightarrow{W^1 \times \cdot} \xrightarrow{\text{tanh}(\cdot)} \xrightarrow{W^2 \times \cdot} \text{score} \]

- Cybenko used
  - The Hahn Banach theorem
  - The Riesz representation theorem
Gradient Descent

• Given a set of examples $((x^t, y^t) \in \mathbb{R}^d \times \mathbb{N}, \ t = 1 \ldots T$, we want to minimize

$$C(\theta) = \sum_{t=1}^{T} c(f_\theta(x^t), y^t)$$

• **Batch** gradient descent

$$\theta \leftarrow \theta - \lambda \frac{\partial C(\theta)}{\partial \theta}$$

★ Update after seeing **all examples**
★ Variants: see your optimization book (Conjugate gradient, BFGS...)
★ Slow in practice

• Take advantage of **redundency**: **stochastic** gradient descent
Pick a random example $t$

$$\theta \leftarrow \theta - \lambda \frac{\partial c(f_\theta(x^t), y^t)}{\partial \theta}$$

★ Update after seeing **one example**
• The learning rate must be chosen **carefully**
• Good idea to use a **validation set**

From (LeCun, 2006)
Consider the network

\[
x \rightarrow w^1 \times \cdot \rightarrow \tanh(\cdot) \rightarrow w^2 \times \cdot \rightarrow \log(1 + e^{-y \cdot})
\]

With one example \((x = 1, y = 1)\) and one hidden unit!

- No progress in some directions
- **Saddle points, plateaux**
Gradient Descent: Tricks Of The Trade

- Initialize properly the weights
  - Not too big: $\tanh(\bullet)$ saturates
  - Not too small: all units would do the same!

- Normalize properly your data (mean/variance)
  - Again, you want to be in the right part of the $\tanh(\bullet)$

- Use a second order approach ($H$ is the Hessian)
  \[
  C(\theta + \epsilon) \approx C(\theta) + \frac{\partial C(\theta)}{\partial \theta} \epsilon + \epsilon^T H(\theta) \epsilon
  \]
  - Costly with full Hessian, consider only the diagonal
  - Estimated on a training subset
  - Be sure it is positive definite!
  - Can be “backpropagated” as the gradient
  - Update with
    \[
    \lambda = \frac{\gamma}{\frac{\partial^2 C}{\partial \theta_k^2} + \mu} \quad \forall k
    \]
In the neural network field: (Rumelhart, Hinton, Williams, 1986)
However, previous possible references exist, including (Leibniz, 1675) and (Newton, 1687)

View the network+loss as a “stack” of layers

\[ x \rightarrow f_1(\bullet) \rightarrow f_2(\bullet) \rightarrow f_3(\bullet) \rightarrow f_4(\bullet) \]

Minimize the score by gradient descent

\[ f(x) = f_L(f_{L-1}(\ldots f_1(x)) \quad \rightarrow \quad \text{How to compute } \frac{\partial f}{\partial w_l} \quad \forall l \quad ?? \]

For e.g., in the Adaline \( L = 2 \)

\[ x \rightarrow w^1 \times \bullet \rightarrow \frac{1}{2}(y - \bullet) \]

\[ \star \quad f_1(x) = w_1 \cdot x \]
\[ \star \quad f_2(f_1) = \frac{1}{2}(y - f_1)^2 \]

\[
\frac{\partial f}{\partial w_1} = \frac{\partial f_2}{\partial f_1} \frac{\partial f_1}{\partial w_1} = y - f_1 \]

\[
\frac{\partial f_1}{\partial w_1} = x
\]

chain rule
Brutal way:

\[
\frac{\partial f}{\partial w^l} = \frac{\partial f_L}{\partial f_{L-1}} \frac{\partial f_{L-1}}{\partial f_{L-2}} \ldots \frac{\partial f_{l+1}}{\partial f_l} \frac{\partial f_l}{\partial w^l}
\]

In the backprop way, each module \( f_l() \)

- **Receive** the gradient w.r.t. its own outputs \( f_l \)
- **Computes** the gradient w.r.t. its own input \( f_{l-1} \) (backward)
- **Computes** the gradient w.r.t. its own parameters \( w^l \) (if any)

\[
\begin{align*}
\frac{\partial f}{\partial f_{l-1}} &= \frac{\partial f}{\partial f_l} \frac{\partial f_l}{\partial f_{l-1}} \\
\frac{\partial f}{\partial w^l} &= \frac{\partial f}{\partial f_l} \frac{\partial f_l}{\partial w^l}
\end{align*}
\]

Often, gradients are efficiently computed using outputs of the module

Do a **forward** before each backward
Examples Of Modules

For simplicity, we denote

- $x$ the input of a module
- $z$ target of a loss module
- $y$ the output of a module $f_l(x)$
- $\tilde{y}$ the gradient w.r.t. the output of each module

<table>
<thead>
<tr>
<th>Module</th>
<th>Forward</th>
<th>Backward</th>
<th>Gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>$y = Wx$</td>
<td>$W^T \tilde{y}$</td>
<td>$\tilde{y} x^T$</td>
</tr>
<tr>
<td>MSE Loss</td>
<td>$y = \frac{1}{2} (x - z)^2$</td>
<td>$x - z$</td>
<td></td>
</tr>
<tr>
<td>Tanh</td>
<td>$y = \tanh(x)$</td>
<td>$\tilde{y} (1 - y^2)$</td>
<td></td>
</tr>
<tr>
<td>Sigmoid</td>
<td>$y = 1/(1 + e^{-x})$</td>
<td>$\tilde{y} (1 - y) y$</td>
<td></td>
</tr>
<tr>
<td>Perceptron Loss</td>
<td>$y = \max(0, -zx)$</td>
<td>$-1_{z \cdot x \leq 0}$</td>
<td></td>
</tr>
</tbody>
</table>

See Lush, Torch5, Theano...
Given a set of examples \((x^t, y^t) \in \mathbb{R}^d \times \mathbb{N}, t = 1 \ldots T\) we want to maximize the (log-)likelihood

\[
\log \prod_{t=1}^{T} p(y^t|x^t) = \sum_{t=1}^{T} \log p(y^t|x^t)
\]

The network outputs a score \(f_y(x)\) per class \(y\)

Interpret scores as conditional probabilities using a softmax:

\[
p(y|x) = \frac{e^{f_y(x)}}{\sum_i e^{f_i(x)}}
\]

In practice we prefer log-probabilities:

\[
\log p(y|x) = f_y(x) - \log \left[ \sum_i e^{f_i(x)} \right]
\]
Assume only two class problems, \( y \in \{-1, +1\} \)

\[
\log p(y = 1|x) = \log \frac{e^{f_1(x)}}{e^{f_1(x)} + e^{f_{-1}(x)}} = -\log(1 + e^{-y(f_1(x) - f_{-1}(x))})
\]

\[
\log p(y = -1|x) = \log \frac{e^{f_{-1}(x)}}{e^{f_1(x)} + e^{f_{-1}(x)}} = -\log(1 + e^{y(f_1(x) - f_{-1}(x))})
\]

Note: only one network output needed

Taking \( z = y(f_1(x) - f_{-1}(x)) \),

\( z \mapsto \log(1 + e^{-z}) \) is a smooth version of SVM cost
Likelihood For Regression

- The target variables \( y \in \mathbb{R} \) are now continuous
- We often consider
  \[
  y | x \sim \mathcal{N}(f(x), \sigma^2)
  \]
- In this case,
  \[
  \log p(y|x) = -\frac{1}{2\sigma^2}||y - f(x)||^2 + \text{cste}
  \]
- Equivalent to Mean Squared Error (MSE) criterion...
- Not great to classification
Unsupervised Training

- How to leverage **unlabeled data** (when there is no $y$)?
- Deep architectures are hard to train: how to **pretrain** each layer?

- “Auto-encoder/bottleneck” network: try to **reconstruct** the input

\[
\begin{align*}
&x \\
&W^1 \circ \tanh(\bullet) \\
&W^2 \circ \tanh(\bullet) \\
&W^3 \\
\end{align*}
\]

- **Caveats:**
  - **PCA** if no $W^2$ layer (Bourlard & Kamp, 1988)
  - It is a **bottleneck** mapping...
Unsupervised Training

Possible improvements:

- No $W^2$ layer, $W^3 = [W^1]^T$ (Bengio et al., 2006)
- Inject noise in $x$, try to reconstruct the true $x$ (Bengio et al., 2008)
- Impose sparsity constraints on the projection (Kavukcuoglu et al., 2008)
A Radial Basis Function (RBF) layer is defined by:

\[
f_{1,i}(x) = e^{-\frac{||x-W_{1,i}||^2}{2\sigma^2}}
\]

Better to find parametrization of \( \sigma \) such that it is strictly positive:

\[
\sigma = \tilde{\sigma} + \theta
\]

Gradient is zero if \( W^1 \) columns are far from training examples

\[\rightarrow \text{Initialize with K-Means}\]
Specialized Layers: 1D Convolutions

Weights are “shared” through time

\[ X = (X_{\bullet 1}, X_{\bullet 2}, \ldots) \quad \text{input (matrix)} \]

\[ W \times \begin{pmatrix} X_{\bullet 1} & X_{\bullet 2} \\ X_{\bullet 2} & X_{\bullet 3} \\ X_{\bullet 3} & X_{\bullet 4} \end{pmatrix} \quad \text{convolution (local embedding for each input column)} \]

Robustness to time shifts:
- Apply sub-sampling (as convolution, but \( W_{\bullet,i} \) contains single value)
- Also called Time Delay Neural Networks (TDNNs)
Same story than in 1D but... in 2D
Specialized Training: Non-Linear CRF

- **Sequence** of $T$ frames $[x]^T$
- The network score for class $k$ at the $t^{th}$ frame is $f([x]^T, k, t, \theta)$
- $A_{kl}$ transition score to jump from class $k$ to class $l$

\[
\begin{align*}
A_{[i]t-1[i]t} + f([x]^T, [i]_t, t, \theta)
\end{align*}
\]

- **Sentence** score for a class label path $[i]^T$

\[
\sum_{t=1}^{T} \left( A_{[i]t-1[i]t} + f([x]^T, [i]_t, t, \theta) \right)
\]

- Conditional likelihood by normalizing w.r.t all possible paths:

\[
\log p([y]^T | [x]^T, \tilde{\theta}) = s([x]^T, [y]^T, \tilde{\theta}) - \logadd s([x]^T, [j]^T, \tilde{\theta}) \quad \forall [j]^T
\]
Normalization computed with recursive **Forward** algorithm:

\[
\delta_t(j) = \text{logAdd}_i \left[ \delta_{t-1}(i) + A_{i,j} + f_\theta(j, x_{1}^T, t) \right]
\]

Termination:

\[
\logadd s([x]_T, [j]_T, \tilde{\theta}) = \text{logAdd}_i \delta_T(i)
\]

Simply **backpropagate through this recursion** with chain rule

- **Non-linear CRFs**: **Graph Transformer Networks** (Bottou et al., 1997)
- Compared to CRFs, we **train features** (network parameters \( \theta \) and transitions scores \( A_{kl} \))

Inference: **Viterbi** algorithm (replace \( \text{logAdd} \) by \( \text{max} \))