Multi-Layer Architectures
Gradient Back-Propagation

Yann LeCun
The Courant Institute of Mathematical Sciences
New York University

http://yann.lecun.com
http://www.cs.nyu.edu/~yann
Non-Linear Models

We can always add “features”, “kernels”, or “basis function” in front of a linear model to make it non-linear.

This is how non-linear SVMs are built.

**Question:** so, why would we need anything else?

**Answer:** the complexity of the real world is very difficult to capture in a kernel.

**How do we solve:**

- The invariance problem in image recognition.
- The structured output problem in sequence labeling (parts of speech tagging, speech recognition, biological sequence analysis....).
Fixed Preprocessing (features, kernels, basis functions)

- Map the inputs into a (higher dimensional) “feature” space
  - With more dimensions, the task is more likely to be linearly separable.

- **Problem:** how should we pick the features so that the task becomes linearly separable in the feature space?
  - **Classical approach 1:** we use our prior knowledge about the problem to hand-craft an appropriate feature set.
  - **Classical Approach 2:** we use a “standard” set of basis functions (RBFs....)
Simplest approach: The Kernel Method (thanks to Wahba's Representer Theorem)

- Make each basis function a “bump” function (a template matcher).
- Place one bump around each training sample.
- Compute a linear combination of the bumps.
- In the “bump space”, we get one separate dimension for each training sample, so if the bumps are narrow enough, we can learn any mapping on the training set.
- To generalize on unseen samples, we adjust the bump widths and we regularize the weights.
- We get a Support Vector Machine.

Problem: an SVM is a glorified template matcher which is only as good as its kernel.
Trainable Front-End, Structured Architectures

The Solutions:

- **Invariance**: Do not use a fixed front-end, **make it trainable**, so it can learn to extract invariant representations.
- **Structure**: Do not use simple linearly-parameterized classifiers, use architectures whose inference process involves multiple non-linear decisions, as well as search and “reasoning”.

We need total flexibility in the design of the architecture of the machine:

- So that we can tailor the architecture to the task
- So that we can build our prior knowledge about the task into the architecture

Multi-Module Architectures.
For Supervised Learning

- We allow the function $F(W,X)$ to be non-linearly parameterized in $W$.
- This allows us to play with a large repertoire of functions with rich class boundaries.
- We assume that $F(W,X)$ is differentiable almost everywhere with respect to $W$. 
Multi-Module Systems: Cascade

- Complex learning machines can be built by assembling Modules into networks.
- A simple example: layered, feed-forward architecture (cascade).
- Computing the output from the input: forward propagation
- Let $X = X_0$, 
  $$X_i = F_i(X_{i-1}, W_i) \quad \forall i \in [1, n]$$
  $$E(Y, X, W) = C(X_n, Y)$$
Object-Oriented Implementation

- Each module is an object (instance of a class).
- Each class has an “fprop” (forward propagation) method that takes the input and output states as arguments and computes the output state from the input state.
- Lush:
  (=> module fprop input output)
- C++:
  module.fprop(input, output);

Y. LeCun: Machine Learning and Pattern Recognition – p. 4/1
Gradient of the Loss, gradient of the Energy

- We assumed early on that the loss depends on $W$ only through the terms $E(W, Y, X^i)$:

$$L(W, Y^i, X^i) = L(Y^i, E(W, 0, X^i), E(W, 1, X^i), \ldots, E(W, k - 1, X^i))$$

- therefore:

$$\frac{\partial L(W, Y^i, X^i)}{\partial W} = \sum_Y \frac{\partial L(W, Y^i, X^i)}{\partial E(W, Y, X^i)} \frac{\partial E(W, Y, X^i)}{\partial W}$$

- We only need to compute the terms $\frac{\partial E(W, Y, X^i)}{\partial W}$

- Question: How do we compute those terms efficiently?
Computing the Gradients in Multi-Layer Systems

To train a multi-module system, we must compute the gradient of $E$ with respect to all the parameters in the system (all the $W_i$).

Let’s consider module $i$ whose fprop method computes $X_i = F_i(X_{i-1}, W_i)$.

Let’s assume that we already know $\frac{\partial E}{\partial X_i}$, in other words, for each component of vector $X_i$ we know how much $E$ would wiggle if we wiggled that component of $X_i$.
Computing the Gradients in Multi-Layer Systems

- We can apply chain rule to compute \( \frac{\partial E}{\partial W_i} \) (how much \( E \) would wiggle if we wiggled each component of \( W_i \)):

\[
\frac{\partial E}{\partial W_i} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}
\]

\[
[1 \times N_w] = [1 \times N_x].[N_x \times N_w]
\]

- \( \frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i} \) is the Jacobian matrix of \( F_i \) with respect to \( W_i \).

\[
\left[\frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}\right]_{kl} = \frac{\partial [F_i(X_{i-1}, W_i)]_k}{\partial [W_i]_l}
\]

- Element \((k, l)\) of the Jacobian indicates how much the \( k \)-th output wiggles when we wiggle the \( l \)-th weight.
Computing the Gradients in Multi-Layer Systems

Using the same trick, we can compute \( \frac{\partial E}{\partial X_{i-1}} \). Let’s assume again that we already know \( \frac{\partial E}{\partial X_i} \), in other words, for each component of vector \( X_i \) we know how much \( E \) would wiggle if we wiggled that component of \( X_i \).

- We can apply chain rule to compute \( \frac{\partial E}{\partial X_{i-1}} \) (how much \( E \) would wiggle if we wiggled each component of \( X_{i-1} \)):

\[
\frac{\partial E}{\partial X_{i-1}} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_{i-1}}
\]

- \( \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_{i-1}} \) is the Jacobian matrix of \( F_i \) with respect to \( X_{i-1} \).
- \( F_i \) has two Jacobian matrices, because it has to arguments.
- Element \((k, l)\) of this Jacobian indicates how much the \( k\)-th output wiggles when we wiggle the \( l\)-th input.
- The equation above is a recurrence equation!
Jacobians and Dimensions

- derivatives with respect to a column vector are line vectors (dimensions: 
  \[ [1 \times N_{i-1}] = [1 \times N_i] \ast [N_i \times N_{i-1}] \])

\[
\frac{\partial E}{\partial X_{i-1}} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_i}
\]

- (dimensions: \[ [1 \times N_{wi}] = [1 \times N_i] \ast [N_i \times N_{wi}] \]):

\[
\frac{\partial E}{\partial W_i} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial W}
\]

- we may prefer to write those equation with column vectors:

\[
\frac{\partial E'}{\partial X_{i-1}} = \frac{\partial F_i(X_{i-1}, W_i)'}{\partial X_{i-1}} \frac{\partial E'}{\partial X_i}
\]

\[
\frac{\partial E'}{\partial W_i} = \frac{\partial F_i(X_{i-1}, W_i)'}{\partial W} \frac{\partial E'}{\partial X_i}
\]
Back-propagation

To compute all the derivatives, we use a backward sweep called the **back-propagation algorithm** that uses the recurrence equation for \( \frac{\partial E}{\partial X_i} \)

\[
\frac{\partial E}{\partial X_n} = \frac{\partial C(X_n, Y)}{\partial X_n}
\]

\[
\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial X_{n-1}}
\]

\[
\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial W_n}
\]

\[
\frac{\partial E}{\partial X_{n-2}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial X_{n-2}}
\]

\[
\frac{\partial E}{\partial W_{n-1}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial W_{n-1}}
\]

...etc, until we reach the first module.

we now have all the \( \frac{\partial E}{\partial W_i} \) for \( i \in [1, n] \).
Object-Oriented Implementation

- Each module is an object (instance of a class).
- Each class has a “bprop” (backward propagation) method that takes the input and output states as arguments and computes the derivative of the energy with respect to the input from the derivative with respect to the output:
  - Lush: \( \text{==> module bprop input output} \)
  - C++: `module.bprop(input, output);`
- the objects input and output contain two slots: one vector for the forward state, and one vector for the backward derivatives.
- the method `bprop` computes the backward derivative slot of `input`, by multiplying the backward derivative slot of `output` by the Jacobian of the module at the forward state of `input`. 
A fully-connected, feed-forward, multi-layer neural nets can be implemented by stacking three types of modules.

- **Linear modules:** $X_{\text{in}}$ and $X_{\text{out}}$ are vectors, and $W$ is a weight matrix.

  $$X_{\text{out}} = W X_{\text{in}}$$

- **Sigmoid modules:**

  $$(X_{\text{out}})_i = \sigma((X_{\text{in}})_i + B_i)$$ where $B$ is a vector of trainable “biases”, and $\sigma$ is a sigmoid function such as tanh or the logistic function.

- **a Euclidean Distance module** $E = \frac{1}{2} ||Y - X_{\text{in}}||^2$. With this energy function, we will use the neural network as a regressor rather than a classifier.
Loss Function

Here, we will use the simple Energy Loss function $L_{\text{energy}}$:

$$L_{\text{energy}}(W, Y^i, X^i) = E(W, Y^i, X^i)$$
OO Implementation: the `state` Class

the internal state of the network will be kept in a “state” class that contains two scalars, vectors, or matrices: (1) the state proper, (2) the derivative of the energy with respect to that state.
Linear Module

The input vector is multiplied by the weight matrix.

- fprop: $X_{out} = WX_{in}$

- bprop to input:
  \[
  \frac{\partial E}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} \frac{\partial X_{out}}{\partial X_{in}} = \frac{\partial E}{\partial X_{out}} W
  \]

- by transposing, we get column vectors:
  \[
  \frac{\partial E'}{\partial X_{in}} = W' \frac{\partial E'}{\partial X_{out}}
  \]

- bprop to weights:
  \[
  \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial X_{outi}} \frac{\partial X_{outi}}{\partial W_{ij}} = X_{inj} \frac{\partial E}{\partial X_{outi}}
  \]

- We can write this as an outer-product:
  \[
  \frac{\partial E'}{\partial W} = \frac{\partial E'}{\partial X_{out}} X_{in}^{\prime}
  \]
Linear Module

Lush implementation:

```
(defclass linear-module object w)
(defmethod linear-module linear-module (nininputs noutputs)
  (setq w (matrix noutputs nininputs)))
(defmethod linear-module fprop (input output)
  (==> output resize (idx-dim :w:x 0))
  (idx-m2dotml :w:x :input:x :output:x) ())
(defmethod linear-module bprop (input output)
  (idx-m2dotml (transpose :w:x) :output:dx :input:dx)
  (idx-m1extml :output:dx :input:x :w:dx) ())
```
Sigmoid Module (tanh: hyperbolic tangent)

- fprop: $(X_{out})_i = \tanh((X_{in})_i + B_i)$
- bprop to input:
  $\left(\frac{\partial E}{\partial X_{in}}\right)_i = \left(\frac{\partial E}{\partial X_{out}}\right)_i \tanh'(\tanh((X_{in})_i + B_i))$
- bprop to bias:
  $\frac{\partial E}{\partial B_i} = \left(\frac{\partial E}{\partial X_{out}}\right)_i \tanh'(\tanh((X_{in})_i + B_i))$
- $\tanh(x) = \frac{2}{1+\exp(-x)} - 1 = \frac{1-\exp(-x)}{1+\exp(-x)}$

(defclass tanh-module object bias)
(defmethod tanh-module tanh-module 1
  (setq bias (apply matrix 1)))
(defmethod tanh-module fprop (input output)
  (==> output resize (idx-dim :bias:x 0))
  (idx-add :input:x :bias:x :output:x)
  (idx-tanh :output:x :output:x))
(defmethod tanh-module bprop (input output)
  (idx-dtanh (idx-add :input:x :bias:x) :input:dx)
  (idx-mul :input:dx :output:dx :input:dx)
  (idx-copy :input:dx :bias:dx) ())

Y. LeCun: Machine Learning and Pattern Recognition – p. 21/2
Euclidean Module

- fprop: $X_{out} = \frac{1}{2} ||X_{in} - Y||^2$
- bprop to $X$ input: $\frac{\partial E}{\partial X_{in}} = X_{in} - Y$
- bprop to $Y$ input: $\frac{\partial E}{\partial Y} = Y - X_{in}$

(defclass euclidean-module object)
(defmethod euclidean-module run (input1 input2 output)
  (idx-copy :input1:x :input2:x)
  (:output:x 0) ())
(defmethod euclidean-module fprop (input1 input2 output)
  (idx-sqrdist :input1:x :input2:x :output:x)
  (:output:x (* 0.5 (:output:x))) ())
(defmethod euclidean-module bprop (input1 input2 output)
  (idx-sub :input1:x :input2:x :input1:dx)
  (idx-dotm0 :input1:dx :output:dx :input1:dx)
  (idx-minus :input1:dx :input2:dx))

Y. LeCun: Machine Learning and Pattern Recognition – p. 22/1
Assembling the Network: A single layer

;; One layer of a neural net
(defclass nn-layer object
  linear ; linear module
  sum ; weighted sums
  sigmoid ; tanh-module
)

(defun nn-layer nn-layer (nininputs noutputs)
  (setq linear (new linear-module nininputs noutputs))
  (setq sum (new state noutputs))
  (setq sigmoid (new tanh-module noutputs)) ()))

(defun nn-layer fprop (input output)
  (=== linear fprop input sum)
  (=== sigmoid fprop sum output) ())

(defun nn-layer bprop (input output)
  (=== sigmoid bprop sum output)
  (=== linear bprop input sum) ())

Y. LeCun: Machine Learning and Pattern Recognition – p. 23/1
Assembling a 2-layer Net

Class implementation for a 2 layer, feed forward neural net.

(defclass nn-2layer object
  (layer1 ; first layer module
    hidden ; hidden state
    layer2 ; second layer
  )
  (defmethod nn-2layer nn-2layer (ninputs nhi
    (setq layer1 (new nn-layer ninputs nhidde
    (setq hidden (new state nhidden))
    (setq layer2 (new nn-layer nhidde noutpu
Assembling the Network: fprop and bprop

Implementation of a 2 layer, feed forward neural net.

(defmethod nn-2layer fprop (input output)
  (==> layer1 fprop input hidden)
  (==> layer2 fprop hidden output) ()))

(defmethod nn-2layer bprop (input output)
  (==> layer2 bprop hidden output)
  (==> layer1 bprop input hidden) ()))
Assembling the Network: training

- A training cycle:
- pick a sample \((X^i, Y^i)\) from the training set.
- call fprop with \((X^i, Y^i)\) and record the error
- call bprop with \((X^i, Y^i)\)
- update all the weights using the gradients obtained above.
- with the implementation above, we would have to go through each and every module to update all the weights. In the future, we will see how to “pool” all the weights and other free parameters in a single vector so they can all be updated at once.
Any Architecture works

- **Any connection is permissible**
  - Networks with loops must be “unfolded in time”.

- **Any module is permissible**
  - As long as it is continuous and differentiable almost everywhere with respect to the parameters, and with respect to non-terminal inputs.
Example: what is the loss function for the simplest 2-layer neural net ever

Function: 1-1-1 neural net. Map 0.5 to 0.5 and -0.5 to -0.5 (identity function) with quadratic cost:

\[ y = \tanh(W_1 \tanh(W_0 x)) \quad L = (0.5 - \tanh(W_1 \tanh(W_0 0.5))^2 \]