Deep Supervised Learning (modular approach)
Complex learning machines can be built by assembling modules into networks.

Simple example: sequential/layered feed-forward architecture (cascade)

Forward Propagation:

\[ X_i = F_i(X_{i-1}, W_i) \quad \forall i \in [1, n] \]

\[ E(Y, X, W) = C(X_n, Y) \]
Multimodule Systems: Implementation

Each module is an object
- Contains trainable parameters
- Inputs are arguments
- Output is returned, but also stored internally
- Example: 2 modules m1, m2

Torch7 (by hand)
- hid = m1:forward(in)
- out = m2:forward(hid)

Torch7 (using the nn.Sequential class)
- model = nn.Sequential()
- model:add(m1)
- model:add(m2)
- out = model:forward(in)
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 Gradient 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 Gradient 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: $[1 \times N_{i-1}] = [1 \times N_i] \times [N_i \times N_{i-1}]$)

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

- (Dimensions: $[1 \times N_{w_i}] = [1 \times N_i] \times [N_i \times N_{w_i}]$):

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

- We may prefer to write those equations with column vectors:

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

\[
\frac{\partial E'}{\partial W_i} = \frac{\partial F_i(X_{i-1}, W_i)}{\partial W} \frac{\partial E'}{\partial X_i}
\]
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 $i \in [1, n]$.
Backpropagation through a module

- Contains trainable parameters
- Inputs are arguments
- Gradient with respect to input is returned.
- Arguments are input and gradient with respect to output

**Torch7 (by hand)**
- `hidg = m2:backward(hid, outg)`
- `ing = m1:backward(in, hidg)`

**Torch7 (using the nn.Sequential class)**
- `ing = model:backward(in, outg)`
The input vector is multiplied by the weight matrix.

- **fprop:** \( X_{\text{out}} = W X_{\text{in}} \)
- **bprop to input:**
  \[
  \frac{\partial E}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} \frac{\partial X_{\text{out}}}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} W
  \]
- **by transposing, we get column vectors:**
  \[
  \frac{\partial E}{\partial X_{\text{in}}} = W' \frac{\partial E}{\partial X_{\text{out}}}
  \]
- **bprop to weights:**
  \[
  \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial X_{\text{outi}}} \frac{\partial X_{\text{outi}}}{\partial W_{ij}} = X_{\text{inj}} \frac{\partial E}{\partial X_{\text{outi}}}
  \]
- **We can write this as an outer-product:**
  \[
  \frac{\partial E'}{\partial W} = \frac{\partial E'}{\partial X_{\text{out}}} X_{\text{in}}'
  \]
Tanh module (or any other pointwise function)

- **fprop:** \((X_{out})_i = \tanh((X_{in})_i + B_i)\)

- **bprop to input:**
  \[
  (\frac{\partial E}{\partial X_{in}})_i = (\frac{\partial E}{\partial X_{out}})_i \tanh'( (X_{in})_i + B_i )
  \]

- **bprop to bias:**
  \[
  \frac{\partial E}{\partial B_i} = (\frac{\partial E}{\partial X_{out}})_i \tanh'( (X_{in})_i + B_i )
  \]

- **tanh:**
  \[
  \tanh(x) = \frac{2}{1+\exp(-x)} - 1 = \frac{1-\exp(-x)}{1+\exp(-x)}
  \]
Euclidean Distance 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: $\frac{\partial E}{\partial Y} = Y - X_{in}$
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.
Torch7 is based on the Lua language
- Simple and lightweight scripting language, dominant in the game industry
- Has a native just-in-time compiler (fast!)
- Has a simple foreign function interface to call C/C++ functions from Lua

Torch7 is an extension of Lua with
- A multidimensional array engine with CUDA and OpenMP backends
- A machine learning library that implements multilayer nets, convolutional nets, unsupervised pre-training, etc
- Various libraries for data/image manipulation and computer vision
- A quickly growing community of users

Single-line installation on Ubuntu and Mac OSX:
- curl -s https://raw.githubusercontent.com/clementfarabet/torchinstall/master/install | bash

Torch7 Machine Learning Tutorial (neural net, convnet, sparse auto-encoder):
Example: building a Neural Net in Torch7

Net for SVHN digit recognition
10 categories
Input is 32x32 RGB (3 channels)
1500 hidden units

Creating a 2-layer net
Make a cascade module
Reshape input to vector
Add Linear module
Add tanh module
Add Linear Module
Add log softmax layer
Create loss function module

Noutputs = 10;
nfeats = 3; Width = 32; height = 32
ninputs = nfeats*width*height
nhiddens = 1500

-- Simple 2-layer neural network
model = nn.Sequential()
model:add(nn.Reshape(ninputs))
model:add(nn.Linear(ninputs,nhiddens))
model:add(nn.Tanh())
model:add(nn.Linear(nhiddens,noutputs))
model:add(nn.LogSoftMax())
criterion = nn.ClassNLLCriterion()

See Torch7 example at http://bit.ly/16tyLAx
Example: Training a Neural Net in Torch7

for t = 1, trainData:size(), batchSize do
  inputs, outputs = getNextBatch()
  local feval = function(x)
    parameters:copy(x)
    gradParameters:zero()
    local f = 0
    for i = 1, #inputs do
      local output = model:forward(inputs[i])
      local err = criterion:forward(output, targets[i])
      f = f + err
      local df_do = criterion:backward(output, targets[i])
      model:backward(inputs[i], df_do)
    end
    gradParameters:div(#inputs)
    f = f/#inputs
    return f, gradParameters
  end
  optim.sgd(feval, parameters, optimState)
end
% F-PROP

for i = 1 : nr_layers - 1
    [h{i}  jac{i}]  =  nonlinearity(W{i} * h{i-1} +  b{i});
end

h{nr_layers-1}  =  W{nr_layers-1} * h{nr_layers-2}  +   b{nr_layers-1};
prediction  =  softmax(h{l-1});

% CROSS ENTROPY LOSS

loss  =  -  sum(sum(log(prediction) .*  target)) / batch_size;

% B-PROP

dh{l-1}  =  prediction  -  target;

for i = nr_layers – 1 : -1 : 1
    Wgrad{i}  =  dh{i} * h{i-1}';
    bgrad{i}  =  sum(dh{i}, 2);
    dh{i-1}  =  (W{i}' * dh{i})  .*  jac{i-1};
end

% UPDATE

for i = 1 : nr_layers - 1
    W{i}  =  W{i}  –  (lr / batch_size)  *  Wgrad{i};
    b{i}  =  b{i}  –  (lr / batch_size)  *  bgrad{i};
end
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:

\[ y = \tanh(W_1 \tanh(W_0 \cdot x)) \quad L = (0.5 - \tanh(W_1 \tanh(W_0 0.5))^2 \]
Use ReLU non-linearities (tanh and logistic are falling out of favor)

Use cross-entropy loss for classification

Use Stochastic Gradient Descent on minibatches

Shuffle the training samples

Normalize the input variables (zero mean, unit variance)

Schedule to decrease the learning rate

Use a bit of L1 or L2 regularization on the weights (or a combination)
  But it's best to turn it on after a couple of epochs

Use “dropout” for regularization

Lots more in [LeCun et al. “Efficient Backprop” 1998]

**SOFTWARE**

<table>
<thead>
<tr>
<th><strong>Torch7: learning library that supports neural net training</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- <a href="http://www.torch.ch">http://www.torch.ch</a></td>
</tr>
<tr>
<td>- <a href="http://eblearn.sf.net">http://eblearn.sf.net</a> (C++ Library with convnet support by P. Sermanet)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Python-based learning library (U. Montreal)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- <a href="http://deeplearning.net/software/theano/">http://deeplearning.net/software/theano/</a> (does automatic differentiation)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>RNN</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- <a href="http://www.fit.vutbr.cz/~imikolov/rnnlm">www.fit.vutbr.cz/~imikolov/rnnlm</a> (language modeling)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CUDAMat &amp; GNumpy</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- code.google.com/p/cudamat</td>
</tr>
<tr>
<td>- <a href="http://www.cs.toronto.edu/~tijmen/gnumpy.html">www.cs.toronto.edu/~tijmen/gnumpy.html</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Misc</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- <a href="http://www.deeplearning.net/software_links">www.deeplearning.net/software_links</a></td>
</tr>
</tbody>
</table>
Convolutional Nets


- see yann.lecun.com/exdb/publis for references on many different kinds of convnets.

- see http://www.cmap.polytechnique.fr/scattering/ for scattering networks (similar to convnets but with less learning and stronger mathematical foundations)
Applications of Convolutional Nets


- Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala and Yann LeCun: Pedestrian Detection with Unsupervised Multi-Stage Feature Learning, CVPR 2013


- Burger, Schuler, Harmeling: Image Denoising: Can Plain Neural Networks Compete with BM3D?, Computer Vision and Pattern Recognition, CVPR 2012,
Applications of RNNs

- Boden “A guide to RNNs and backpropagation” Tech Report 2002
- Graves “Speech recognition with deep recurrent neural networks” ICASSP 2013
Deep Learning & Energy-Based Models


Practical guide


– L. Bottou, Stochastic gradient descent tricks, Neural Networks, Tricks of the Trade Reloaded, LNCS 2012.

– Y. Bengio, Practical recommendations for gradient-based training of deep architectures, ArXiv 2012